FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Eclipse IDE for the programming languages of the standard IEC 61131

(IDE Eclipse para as linguagens de programação da norma IEC 61131)

Filipe Miguel de Jesus Ramos

Dissertação Realizada no Âmbito do
Mestrado em Engenharia Electrotécnica e de Computadores
Major Telecomunicações

Orientador: Mário Jorge Rodrigues de Sousa (Prof. Dr.)

24 de julho de 2014

U.PORTO
FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

MIEEC - MESTRADO INTEGRADO EM ENGENHARIA ELETROTÉCNICA E DE COMPUTADORES | **2013/2014**

A Dissertação intitulada

"Eclipse IDE for the Programming Languages of the Standard IEC 61131"

foi aprovada em provas realizadas em 24-07-2014

o júri

Presidente Professor Doutor Rui Manuel Esteves Araújo
Professor Auxiliar do Departamento de Engenharia Eletrotécnica e de Computadores da Faculdade de Engenharia da Universidade do Porto

Professor Doutor Jaime Francisco Cuz Fonseca
Professor Associado do Departamento de Electrónica Industrial da Escola de Engenharia da Universidade do Minho

Professor Doutor Mário Jorge Rodrigues de Sousa
Professor Auxiliar do Departamento de Engenharia Eletrotécnica e de Computadores da Faculdade de Engenharia da Universidade do Porto

O autor declara que a presente dissertação (ou relatório de projeto) é da sua exclusiva autoria e foi escrita sem qualquer apoio externo não explicitamente autorizado. Os resultados, ideias, parágrafos, ou outros extratos tomados de ou inspirados em trabalhos de outros autores, e demais referências bibliográficas usadas, são corretamente citados.

Autor - Filipe Miguel de Jesus Ramos

Faculdade de Engenharia da Universidade do Porto

Abstract

This dissertation aims to present the development of an IDE for the programming languages of the norm IEC 61131. The IDE should be an integral part of the Eclipse development environment.

In the present paper are presented the various steps taken during the development of the main objectives: the construction of textual editors for the text-based languages, and of graphic editors for Sequential Function Chart.

A preliminary study was made of the IDE's already on the market, whether commercial or open source, in order to better understand the main aspects that the user seeks in this type of IDE. It was also made a preliminary study of the Eclipse API in order to understand how to proceed with the development of components that interconnect the Eclipse environment.

It were then constructed two text editors and one graphic editor for the languages Instruction List, Structured Text, and Sequential Function Chart, respectively. The textual editors are based on the Eclipse editors architecture, and the graphic editor is based on the Graphiti framework, which works on top of the Eclipse graphical editors architecture.

Resumo

A presente dissertação tem como objetivo o desenvolvimento de um IDE para as linguagens de programação da norma IEC 61131. Esse IDE deve ser parte integrante do ambiente de desenvolvimento Eclipse.

No presente documento são apresentados os vários passos dados durante o desenvolvimento dos objetivos principais: a construção de editores textuais para as linguagens baseadas em texto, e a construção de editores gráficos para a linguagem *Sequential Function Chart*.

Foi inicialmente feito um estudo dos IDE's presentes no mercado, quer comerciais, quer *open source*, de forma a melhor perceber quais os principais aspetos que o utilizador procura num IDE deste tipo. Foi também feito um estudo preliminar da API Eclipse, por forma a perceber como se procede ao desenvolvimento de componentes que se interliguem no ambiente Eclipse.

Foram então construídos dois editores textuais e um editor gráfico para as linguagens *Instruction List*, *Structured Text* e *Sequential Function Chart*, respetivamente. Os editores textuais são baseados na arquitetura de editores do Eclipse, e o editor gráfico é baseado na *framework* Graphiti, que trabalha por cima da arquitetura de editores gráficos do Eclipse.

Conteúdo

Lista de Figuras

Abreviaturas e Símbolos

ANSI	American National Standards Institute
API	Application Programming Interface
CLP	Controladores Lógicos Programáveis
CP	Controladores Programáveis
DTA	Diagram Type Agent
EMF	Eclipse Modeling Framework
FB	Function Block
FBD	Function Block Diagram
GEF	Graphical Editing Framework
HMI	Human Machine Interface
IDE	Integrated Development Environment
IEC	International Electrotechnical Commission
IL	Instruction List
LD	Ladder Diagram
MVC	Model-View-Controller
PC	Personal Computer
PDE	Plug-in Development Environment
PLC	Programmable Logic Controller
POU	Program Organization Unit
SAMA	Scientific Apparatus Makers Association
SFC	Sequential Function Chart
ST	Structured Text
SWT	Standard Widget Toolkit
UDT	User Defined Types
UML	Unified Modeling Language
XMI	XML Metadata Interchange
XML	eXtensible Markup Language

Capítulo 1

Contexto

1.1 Controladores Lógicos Programáveis

Um Controlador Lógico Programável (PLC) é um computador digital usado em aplicações industriais para automação de sistemas eletromecânicos (*e.g.*, linhas de montagem automóvel [1]), construído por forma a suportar condições adversas, dificilmente suportáveis por um computador pessoal (*e.g.*, ruído elétrico, vibrações e impactos) — ver Figura 1.1. A programação dos PLC's é feita pelo utilizador, e os programas ficam guardados numa memória não volátil. Recebe nas entradas sinais de sensores que são logicamente combinados de forma a gerar sinais de saída para controlar atuadores. Atualmente os PLC's podem ser ligados a uma rede e programados remotamente [2].

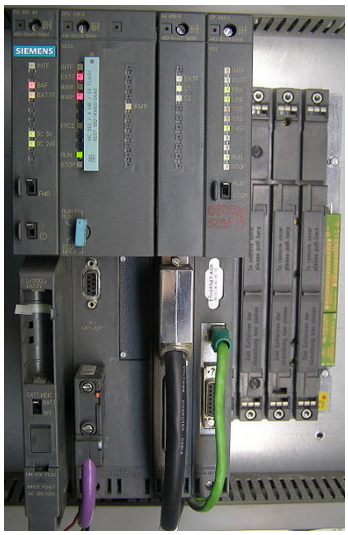


Figura 1.1: Exemplo de um PLC (Siemens Simatic S7 400) [3].

De forma a diminuir o esforço dos programadores e aumentar a interoperabilidade dos mesmos (e respetivos programas), foi preciso padronizar protocolos e linguagens de programação para os PLC's. Foi nesse sentido que em 1990 a International Electrotechnical Commission criou a norma IEC 61131 [4].

Embora tenha sido lançada recentemente a terceira edição da norma (fevereiro de 2013), no nosso projeto vamos continuar a referir a segunda edição, pois as alterações introduzidas não se enquadram no âmbito do projeto.

1.2 A norma IEC 61131

A norma IEC 61131 está dividida em nove partes, cada uma sumariza os requisitos de *hardware* e de programação de PLC's modernos, estabelecendo regras que vão desde características electro-mecânicas dos PLC's, até à definição das comunicações entre eles. Mais especificamente, a parte três da norma (IEC 61131-3, publicada pela primeira vez em dezembro de 1993) é um guia para a programação de PLC's, unificando as várias linguagens de programação utilizadas por todo o mundo, e orientando-as para uma evolução futura [5].

As nove partes que formam a norma IEC 61131 são:

- Parte 1: Informações gerais;
- Parte 2: Testes e requisitos do equipamento;
- Parte 3: Linguagens de programação;
- Parte 4: Diretrizes de utilização;
- Parte 5: Comunicações;
- Parte 6: Segurança funcional;
- Parte 7: Programação de controlo difuso;
- Parte 8: Directrizes para a aplicação e implementação das linguagens de programação;
- Parte 9: Interface de comunicação digital Single-Drop para pequenos sensores e atuadores.

Na terceira parte da norma são definidas a semântica e sintaxe de duas linguagens de programação textuais: *Structured Text* (ST) e *Instruction List* (IL); e três gráficas: *Function Block Diagram* (FBD), *Ladder Diagram* (LD) e *Sequential Function Chart*[1] (SFC) [7]. Na Secção 2.1.2 serão abordadas com maior detalhe. Com estas definições a norma pretende fornecer ao utilizador a linguagem mais adequada para qualquer problema: quer necessite de linguagens textuais ou gráficas, de baixo ou alto nível.

Embora tenha demorado algum tempo, a norma acabou por ser largamente adotada, possibilitando a diferentes programadores de diferentes empresas a combinação de componentes de programas e a reutilização de software de alto nível [8].

[1] A SFC não é estritamente uma linguagem de programação, uma vez que nenhum programa pode ser escrito em SFC sem que sejam usados blocos de uma ou mais das outras linguagens de programação. Serve primeiramente para definir máquinas de estado [6]. No entanto, por questões de simplicidade vamos continuar a tratá-la como se o fosse uma linguagem gráficas, sempre que daí não resulte qualquer desvantagem.

1.3 Ferramentas *Open Source* Existentes

De modo a facilitar a escrita de programas foram criados vários IDE's, alguns deles *Open Source*. Os mais usados desta categoria são o *4DIAC* e o *Beremiz*.

Estes programas vieram preencher um importante espaço vazio, mas nenhum deles permite uma utilização satisfatória. Os editores das linguagens gráficas contêm bastantes lacunas, muitas das vezes dificultando o desenho e a legibilidade dos programas. Durante o desenvolvimento de grandes projetos, com programas longos e complexos, os IDE's tornam-se lentos e não responsivos.

O programa a desenvolver poderia basear-se em um ou mais dos IDE's *Open Source* existentes, mas dados os problemas atrás mencionados, optou-se pela programação de um IDE de raiz.

1.4 Objetivos

A presente dissertação tem como objetivo a elaboração de um IDE *Open Source* para linguagens da norma IEC 61131. Esse IDE deve ser incorporado na ferramenta de desenvolvimento Eclipse, e deve conter dois editores textuais, um para ST e outro para IL, e um editor gráfico para SFC. Descreve-se de seguida cada um desses objetivos.

1.4.1 Editores Textuais

O desenvolvimento de projetos nas linguagens textuais é possível através da utilização de dois editores textuais criados para esse efeito. Foram criados editores textuais para as linguagens *Instruction List* e *Structured Text*.

1.4.1.1 Syntax Highlight

Uma das características que queríamos ver nos editores textuais era a *syntax highlight*. A *syntax highlight* permite colorir de forma diferente pedaços de texto com significados distintos: comentários, variáveis, *etc.* (ver Figura 1.2). É uma das principais características de um editor profissional, permitindo uma melhor leitura do código, e facilitando a deteção de erros. Para implementar esta funcionalidade tivemos de compreender melhor o funcionamento interno dos editores textuais, nomeadamente o funcionamento do particionamento do documento (ver Secção 3.1.2) e da maneira como o Eclipse processa as alterações que vão sendo feitas ao texto do editor pelo utilizador (ver Secção 3.1.1).

1.4.2 Editores Gráficos

Para permitir a programação em SFC foi necessário criar um editor gráfico. Uma característica que queríamos ter no nosso editor era a possibilidade de inserir e conectar todos os tipos de blocos da linguagem gráfica, de uma forma fácil e intuitiva, respeitando as regras imposta pela norma

```
(* Comentário... *)

FUNCTION_BLOCK R_TRIG
    CONST
        PI : 3.141_593_6;
    END_CONST

    VAR
        S : STRING := "String";
        E : BYTE := 0;
        C : CHAR := '9';
    END_VAR
    IF D < 0.0 THEN
        NROOTS := 0;
    ELSIF D = 0.0 THEN
        NROOTS := 1;
        X1 := - B/(2.0*A);
    ELSE
        NROOTS := 2;
        X1 := (- B + SQRT(D))/(2.0*A);
        X2 := (- B - SQRT(D))/(2.0*A);
    END_IF;
```

Figura 1.2: Syntax Highlight do nosso IDE.

IEC 61131. Para criarmos este editor tivemos de compreender melhor a criação de metamodelos (ver Secção 3.2.1), bem como as ferramenta da programação gráfica do Eclipse (ver Secção 3.2.2).

1.4.3 Gravação em XML

Um outra objetivo do nosso projeto era a gravação dos programas, criados nos nossos editores, para *PLCopen XML* (ver Secção 2.5). A conversão dos programas em ST e IL para *PLCopen XML* foi realizada, e também no editor gráfico para SFC implementámos a possibilidade de o utilizador fazer essa conversão. Em ambos os casos, para converter um programa para XML basta ao utilizador escolher a devida opção num menu criado para o efeito. É assim possível abrir um programa criado nos nossos editores, noutros IDE's que suportem a importação de ficheiros XML no formato *PLCopen XML*.

1.4.4 Integração com o Compilador MATIEC

Por último, tinhamos também como objetivo a possibilidade de os programas criados com os nossos editores serem compilados com MATIEC. Para isso eles têm de estar representados em texto simples. Os projetos criados em *Instruction List* e *Structured Text* são gravados pelo nosso editor na sua forma textual. Dessa forma ficam já preparados para serem compilados pelo MATIEC. Já os projetos em *Sequential Function Chart* têm de ser convertidos pelo nosso editor em ficheiros de texto compatíveis com a norma IEC 61131-3, para ficarem prontos a ser compilados com o compilador MATIEC. Essa conversão é feita quando o utilizador escolhe a devida opção num menu criado para o efeito.

Capítulo 2

Estado da Arte

2.1 A norma IEC 61131-3

A norma IEC 61131-3 é a terceira parte (de nove) do *standart* IEC 61131. No contexto desta norma, as funções, grupos de funções e os programas são chamados de Unidades de Organização de Programas (*Program Organization Units* — POU). São definidas na norma as cinco linguagens de programação, os tipos de dados que podem ser usados na escrita dos programas, e vários elementos de configuração. Segue-se uma breve descrição de cada.

2.1.1 POU's

A norma IEC 61131-3 incentiva o desenvolvimento estruturado de programas permitindo que um programa seja dividido em elementos funcionais. Esses elementos são a menor unidade de software independente de um programa — os POU's. Estes podem ser de três tipos: Programas (PROG), Funções (FUN) e Blocos de Funções (FB). Os POU's podem ser reutilizados, e podem ser escritos em mais do que uma linguagem. Segundo as definições da norma, um POU é composto por um cabeçalho e pelo corpo: no cabeçalho são definidas as variáveis, e no corpo encontram-se as instruções para execução [5].

O POU de mais alto nível é o Programa. Pode ser escrito em qualquer das linguagens da norma, e pode conter Funções e Blocos de Funções, mas não outros Programas.

Uma Função é um bloco de código que pode ser chamado em várias partes de um programa, à imagem do que acontece com as funções da linguagem C, mas ao contrário destas, não permite recursão, e é mais limitada no que respeita aos parâmetros de entrada e saída. Não podem ser escritas em SFC.

Um FB é um bloco de código que se pretende usar várias vezes ao longo do programa. Ao agrupar esse código num FB não há necessidade de o repetir. É muito parecido com uma Função, mas com propriedades adicionais: pode conter variáveis persistentes e pode ser escrito em SFC. Para além disso existe uma diferença fundamental: ao passo que as funções são chamadas diretamente, um FB tem de ser instanciado primeiro. Comporta-se como uma classe de uma linguagem de programação de alto nível (embora com algumas limitações), permitindo criar estruturas de

dados complexas, e abrindo portas à programação orientada a objectos. Podem conter variáveis de entrada, saída e entrada/saída, bem como variáveis internas persistentes, e voláteis, e é permitido aceder diretamente a uma variável de saída ou entrada de um FB sem o invocar. Um FB não pode ser instanciado dentro de uma Função.

Cada POU começa com uma *keyword* que identifica o seu tipo (*e.g.*, FUNCTION_BLOCK). Segue-se o cabeçalho, onde se declaram as variáveis, agrupadas por blocos de diferentes tipos, cada bloco identificado por uma *keyword* específica (ver Secção 2.1.7). De seguida vem o corpo do POU, onde é feita a programação. Finalmente o POU termina com a *keyword* complementar daquela com que começou (*e.g.*, END_FUNCTION_BLOCK).

2.1.2 Structured Text

A ST é uma linguagem de alto nível, sintaticamente parecida com *Pascal*[1] (ISO7185). É estruturada por blocos e suporta declarações complexas como *IF...THEN...ELSE* e *WHILE...DO*. Das linguagens da norma, a ST é a mais flexível, adaptando-se bem a qualquer tipo de situação.

Exemplo:

```
Txt := TXTS[Estado];
CASE Estado OF
   1: Fechar();
ELSE
   Erro();
END_CASE;
```

2.1.3 Instruction List

Ao contrário da ST, a IL é uma linguagem de baixo nível, muito parecida com *Assembly*. As instruções são executadas uma por linha, e podem ser usadas chamadas a funções e *jumps* para controlo de fluxo. É uma linguagem pouco usada, e bastante limitada. É usada principalmente em pequenos programas, ou em programas que necessitem de grande otimização.

Exemplo:

```
       LD     Velocidade
       GT     120
       JMPCN  ACE_OK
       LD     Acel
ACE_OK LD     1
       ST     %Q75
```

[1] De facto, é baseada em *Pascal*.

2.1.4 Function Block Diagram

Tal como o nome indica, um FBD é um diagrama de blocos que descreve uma função (ou a instância de um FB) entre variáveis de entrada e de saída. Usa blocos elementares (que representam funções ou instâncias de um FB) conectados por linhas orientadas que transportam os dados. Em qualquer bloco as entradas encontram-se sempre do lado esquerdo e as saídas do lado direito. Um diagrama exemplo pode ser visto na Figura 2.1. O FBD permite criar diagramas fáceis de interpretar, e têm a vantagem de destacar o fluxo de informação e o processamento de sinais.

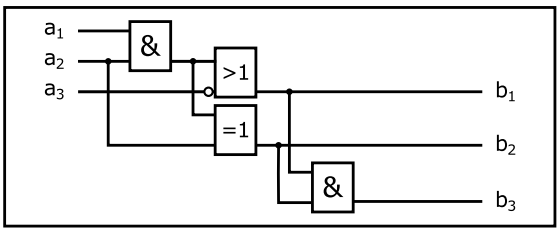


Figura 2.1: Exemplo de um programa em *FBD* [9, adaptada].

2.1.5 Ladder Diagram

A linguagem LD tem uma semelhança aparente com diagramas de circuitos elétricos com relés. Usa interruptores e bobines inseridos num percurso onde circula corrente elétrica, da esquerda para a direita. Tem duas colunas verticais, uma de cada lado, onde se podem ligar os circuitos. Cada bobine do LD representa um bit no PLC. Um contacto pode representar um interruptor real, ou um estado do PLC. Um exemplo de um programa em LD pode ser visto na Figura 2.2. É largamente usada para escrita de programas que se foquem no controlo lógico, sequencial ou discreto.

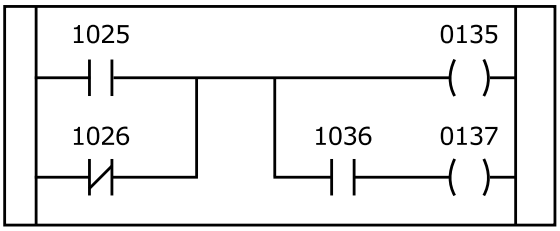


Figura 2.2: Exemplo de um programa em *LD* [9, adaptada].

2.1.6 Sequential Function Chart

Os diagramas SFC são baseados na linguagem GRAFCET — um tipo particular de máquina de estados. Uma máquina de estados é um modelo de computação matemático usado no desenho de programas de computador e de circuitos lógicos sequenciais. Representa uma máquina que, num dado momento, apenas pode estar num de vários estados possíveis (chamado de estado atual).

No decorrer do tempo a máquina pode passar de um estado para outro quando ocorrer um evento predefinido, ou quando for verdadeira certa condição. À passagem de um estado para outro chama-se transição. Pode ver-se um exemplo de uma máquina de estados na Figura 2.3: a seta com um círculo azul indica o estado inicial; os restantes círculos representam os estados do sistema; as restantes setas representam transições — a cada transição está associada uma condição.

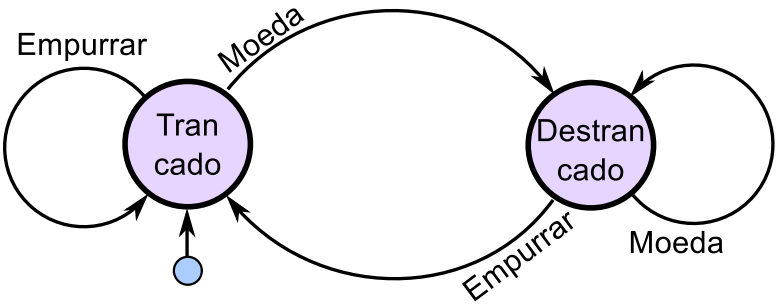


Figura 2.3: Máquina de Estados de um torniquete [10, adaptada].

No caso dos diagramas SFC, aos estados chamam-se passos. Neste documento vamos usar os nomes na língua inglesa, tal como aparecem normalmente na literatura: *steps* e *transitions*. Pode ver-se na Figura 2.4 um diagrama SFC semelhante ao diagrama de Máquinas de Estado da Figura 2.3: O *step* inicial é representado por um retângulo com linha dupla, e os restantes *steps* por um retângulo com linha simples; cada transição é representadas por um linha horizontal.

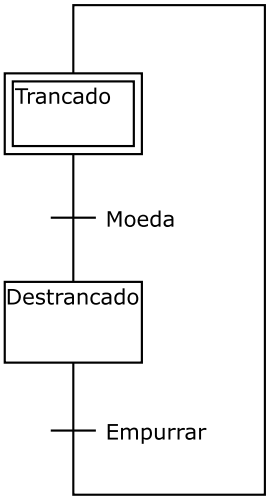


Figura 2.4: Diagrama SFC de um torniquete.

A cada *step* de um diagrama SFC estão associadas ações que se executam quando ele fica ativo (quando se torna no estado atual, utilizando a linguagem das máquinas de estado). As ações podem ser escritas numa qualquer das linguagens da norma.

De modo a acompanhar a execução de programa em SFC a leitura do diagrama deve ser feita de cima para baixo, sempre que tal seja possível.

Para simplificar o desenho do diagramas existem mais alguns elementos gráficos para além dos *steps* e *transitions*. Esses elementos permitem desenhar divergências em ramos paralelos que se executam simultaneamente, ou em ramos paralelos dos quais apenas um pode ser executado. Existem também elementos que permitem fazer saltar a execução do programa entre diferentes pontos do diagrama.

A linguagem SFC é adequada para a estruturação de Programas e Blocos de Funções, controlo de estados e tomadas de decisão. Um exemplo de um programa em SFC pode ser visto na Figura 2.5.

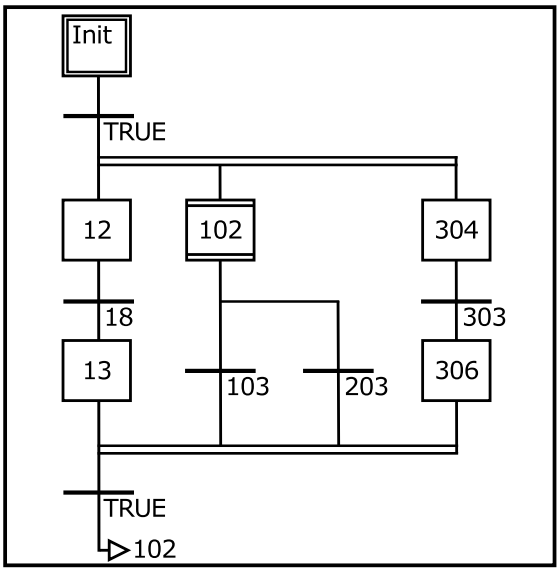


Figura 2.5: Exemplo de um programa em *SFC* [11, adaptada].

Como foi para esta linguagem que escolhemos criar o nosso editor gráfico, vamos dedicar-lhe um pouco mais de atenção. Os elementos básicos da linguagem SFC são:

- Step;
- Transition;
- Action;
- Selection Divergence;
- Selection Convergence;
- Simultaneous Divergence;
- Simultaneous Convergence;
- Jump Step.

Um *Step* representa um estado do programa. É quando a execução do programa chega a um determinado *Step* que são executadas as *Actions* que lhe estão associadas. Para além dos *Steps* normais (*Regular Steps*), existem também os *Steps* iniciais (*Initial Steps*), que marcam o início da execução do programa. Graficamente um *Step* é representado por um retângulo desenhado com linha simples, se for um *Regular Step*, ou com linha dupla, se for um *Initial Step*. Dentro do retângulo que representa o *Step* deve estar o nome do mesmo, que deve ser único dentro do POU onde ele se encontra. Existe ainda um outro tipo de *Step*, chamado *Macro Step*, que não é mais que o encapsulamento de uma sequência SFC num único bloco. Um *Macro Step* não é um sub-programa, mas apenas uma forma de melhorar visualmente a apresentação de um diagrama SFC. A sequência que está a ser encapsulada deve começar e acabar com um *Step*, e o nome do *Macro Step* deve ser o mesmo do primeiro *Step* da sequência encapsulada. Graficamente o retângulo que representa um *Macro Step* tem linhas simples nos lados, e linhas duplas no topo e no fundo (ver Figura 2.6).

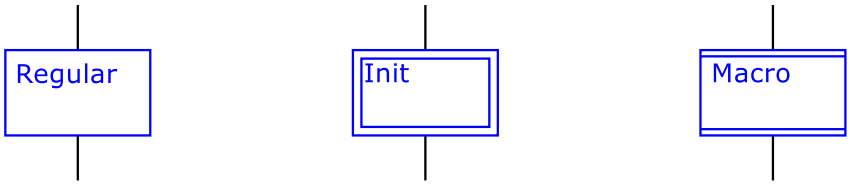


Figura 2.6: *Regular Step*, *Initial Step* e *Macro Step*, com respetivas conexões.

Uma *Transition* representa a condição necessária para que o fluxo do programa possa continuar para o próximo *Step* ou *Steps*. Essa condição tem de ser avaliada como verdadeira ou falsa. As condições podem ser escritas em todas as linguagens da norma, com a excepção de SFC. Uma *Transition* é representada graficamente por uma linha curta horizontal, situada na conexão entre *Steps*, e junto da qual se encontra a condição ou nome da referência indireta para a condição que deve ser satisfeita para que a execução do programa continue por esse ramo (ver Figura 2.7).

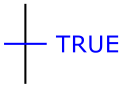


Figura 2.7: *Transition*, com respetivas conexões.

Uma *Action* contém um série de instruções que vão ser executadas quando o fluxo do programa chegar ao *Step* ao qual ela está associada. Podem associar-se várias *Actions* ao mesmo *Step*. As instruções podem ser escritas em qualquer das linguagens definidas na norma IEC 61131-3, e podem ser guardadas por forma a serem referenciadas em várias *Action*. Cada *Action* tem um qualificador que define o ponto de execução ou o tempo de execução da mesma em relação ao *Step*. Os qualificadores são representados por uma ou mais letras maiúsculas. Cada *Action* pode ter um dos seguintes qualificadores:

- Non-Stored (**N**): a *Action* é executada quando o *Step* fica ativo;
- Set (**S**): a *Action* é executada no intervalo de tempo entre o *Step* ficar ativo e o qualificador **R** ser executado;
- Overriding Reset (**R**): as *Actions* executadas com qualificador **S**, **SD**, **DS** e **SL** são abortadas;
- Time Limited (**L**): a *Action* é executada por um período de tempo predefinido, assim que o *Step* fica ativo, mas é abortada se não terminar quando o *Step* ficar inativo;
- Time Delayed (**D**): a *Action* é executada depois de decorrido um período predefinido de tempo contado desde a ativação do *Step*, e fica em execução enquanto o *Step* estiver ativo;
- Pulse (**P**): Quando o *Step* fica ativo, a *Action* é executada durante um ciclo;
- Stored & Time Dalayed (**SD**): a *Action* é executada depois de decorrido um período predefinido de tempo contado desde a ativação do *Step*, e fica em execução mesmo depois do *Step* ficar inativo, até ser executada um qualificador **R**;
- Delayed & Stored (**DS**): a *Action* é executada depois de decorrido um período predefinido de tempo contado desde a ativação do *Step*, e fica em execução mesmo depois do *Step* ficar desativado, até ser executada um qualificador **R**. Ao contrário do qualificador **SD**, no **DS** a *Action* não é executada se o *Step* ficar inativo antes de decorrido o tempo de *delay*.
- Stored & Time Limited (**SL**): a *Action* é executada por um período de tempo predefinido, assim que o *Step* fica ativo, mas é abortada se ocorrer um qualificador **R**.

Graficamente uma Action é representada por dois retângulos desenhados lado a lado. No primeiro retângulo deve aparecer a letra ou letras que representam o qualificador da Action, e no segundo retângulo as instruções a serem executadas, ou o nome do POU onde as instruções estão definidas (ver Figura 2.8).

Figura 2.8: *Action*, com respetiva conexão.

As várias ramificações podem ser usadas para ligar ramos que devem ser executados paralelamente (*Simultaneous Divergence* e *Simultaneous Convergence*) ou para ligar ramos dos quais apenas um deve ser executado (*Selection Divergence* e *Selection Convergence*). Graficamente as *Selection Divergence/Convergence* são representadas por uma linha simples comprida e horizontal, e as *Simultaneous Divergence/Convergence* por uma linha dupla comprida e horizontal (ver Figuras 2.9 e 2.10).

Um *Jump Step* serve para fazer saltar a execução de um ponto do diagrama para outro. Pode colocar-se no fim do diagrama, ou no fim de um ramo de seleção. Não pode ser direcionado para

Figura 2.9: *Selection Convergence* e *Selection Divergence* com respetivas conexões.

um ramo paralelo, a não ser que o próprio *Jump Step* esteja nesse ramo. Graficamente um *Jump Step* é representado por uma seta direcionada para baixo, à frente da qual deve estar o nome que referencia o local para onde o *Jump Step* direciona (ver Figura 2.11).

Quando criamos um diagrama SFC devemos ter sempre uma regra em mente: ignorando todos os restantes objetos, depois de um *Step* deve sempre aparecer uma *Transition*, e depois de uma *Transition* deve sempre aparecer um *Step*.

Na norma IEC 61131 está definida a possibilidade de conversão dos diagramas SFC para texto. Muita da informação referente aos objetos gráficos não é retida na conversão (*e.g.*, tamanho e posição dos elementos), mas a informação relativa à lógica está toda presente. O ficheiro resultante pode ser compilado pelo compilador MATIEC (ver Secção 2.4).

2.1.7 Tipos de Variáveis

Na norma IEC 61131-3 são definidos vários tipos de variáveis. Os principais tipos são: variáveis de entrada, variáveis de saída e variáveis de entrada/saída, bem como variáveis internas, quer persistentes, quer voláteis.

As variáveis internas existem apenas dentro de um POU, e não são usadas fora desse POU. As variáveis de entrada são alimentadas por variáveis de saída presentes noutro POU. Podem ser persistentes (declaradas com a palavra chave VAR), ou voláteis (declaradas com a palavra chave VAR_TEMP). As variáveis de saída (declaradas com a palavra chave VAR_OUTPUT), servem para alimentar variáveis de entrada de outros POU's. As variáveis de entrada/saída (declaradas com a palavra chave VAR_IN_OUT), são fornecidas por outros POU's, como as variáveis de entrada, e são fornecidas a outros POU's, como as variáveis de saída.

2.1.8 Tipos de Dados

A norma IEC 61131-3 define vários tipos de dados para uso na programação dos diferentes POU's. Segue-se uma lista dos principais tipos:

- **Inteiros**: permite vários tipos de inteiros, desde o *Short Integer* (SINT) de 8 bit, ao *Unsigned Long Integer* (ULINT) de 64 bit;
- **Reais**: *Real* (REAL), de 32 bit, ou *Long Real* (LREAL) de 64 bit;
- **Tipos orientados ao Bit**: desde o *Boolean* (BOOL) de 1 bit, ao *Long Word* (LWORD) de 64 bit;

Figura 2.10: *Simultaneous Convergence* e *Simultaneous Divergence* com respetivas conexões.

- **Strings**: *String* (STRING) com carateres de 8 bit, ou *Wide String* (WSTRING), com carateres de 16 bit;
- **Tempo e Data**: permite vários formatos, como TIME (durações temporais) ou DATE (dia, mês e ano);
- **Sub Ranges**: define um intervalo de valores para dada variável;
- **Tipos de dados derivados**: permite criar novos tipos de dados baseados nos anteriores.
  - **Vetores**: contêm vários elementos de um qualquer tipo de dados dos anteriores;
  - **Estruturas**: agrupamentos de vários outros tipos de dados;
  - **Enumerações**: variáveis que pode tomar apenas os valores enumerados;

2.2 IDE's para IEC 61131-3

O mundo dos PLC's pode ser dividido em dois: de um lado existem os PLC's físicos, (tais como os *Allen Bradley*, *Siemens* (ver Figura 1.1), *Omron*, *Schneider*, *Faruc*, *etc.*), do outro lado os *softPLC*, que não são mais que um *software* que corre em PC's e que permite emular as funcionalidades de um PLC físico. Alguns dos *softPLC* existentes são o *SoftPLC*, o *CoDeSys* e o *ISaGRAF*.

Dos IDE's que permitem a programação de softPLC's existem vários compatíveis com as linguagens IEC 61131-3. Alguns são *open source*, outros comerciais. Para a elaboração deste documento foram testados os principais de ambas as categorias. Segue-se uma breve descrição de cada um, indicando quais os seus pontos fortes e fracos.

2.2.1 IDE's Open Source

2.2.1.1 Beremiz

O Beremiz é o resultado de um esforço conjunto da empresa LOLITECH, sediada em Saint-Dié-des-Vosges, França, com a Universidade do Porto. Tem como base, entre outros, o compilador *MATIEC* e o editor gráfico *PLCopen editor* [12]. (Ver Figura 2.12).

O facto de ser programado em *Python* pode diminuir a sua performance em algumas circunstancias, potência alguma dificuldade na sua manutenção e possibilita o aparecimento de *bugs* facilmente evitáveis noutras linguagens (principalmente relacionados com a declaração de variáveis).

Figura 2.11: *Jump Step*, com respetiva conexão.

Ao editor gráfico, também escrito em *Python* faltam algumas características básicas (*e.g.*, devido à dificuldade em alinhar blocos o resultado é visualmente desagradável).

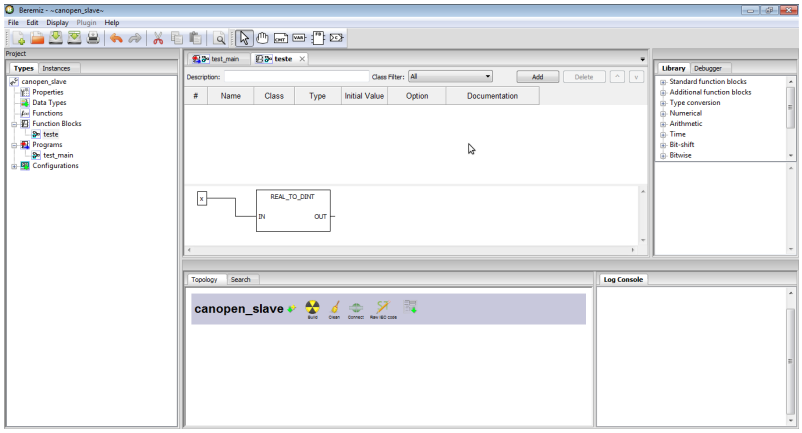


Figura 2.12: Janela do Beremiz.

2.2.1.2 4DIAC

O IDE 4DIAC tem como base a *framework* Eclipse (ver Secção 2.3), e tem como objetivo principal a criação de aplicações compatíveis com a norma IEC 61499 [13]. Pode ver o software na Figura 2.14.

A norma IEC 61499 centra-se nos seus FB's muito específicos. Como se pode ver na Figura 2.13, cada FB da norma tem dois grupos distintos de entradas, e dois grupos distintos de saídas: entradas de eventos, entradas de dados, saídas de eventos e saídas de dados. Quando um evento é ativado na entrada, o código do programa correspondente é executado, tirando partido das entradas de dados associadas ao evento. O programa, depois de executado, coloca dados nas respetivas saídas de dados, e o evento correspondente é ativado na saída.

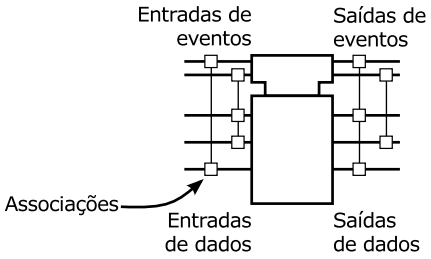


Figura 2.13: Um *Function Block* na norma IEC 61499 [14, adaptada].

Não tendo como foco principal a programação nas linguagens IEC 61131-3, não se mostra verdadeiramente como uma alternativa ao *software* pretendido, e embora fosse possível expandir o IDE 4DIAC de modo a suportar as linguagens IEC 61131-3, optámos por criar um IDE independente, mas tendo em vista uma possível integração futura, dado que os FB do IEC 61499 podem ser programados usando as linguagens IEC 61131-3.

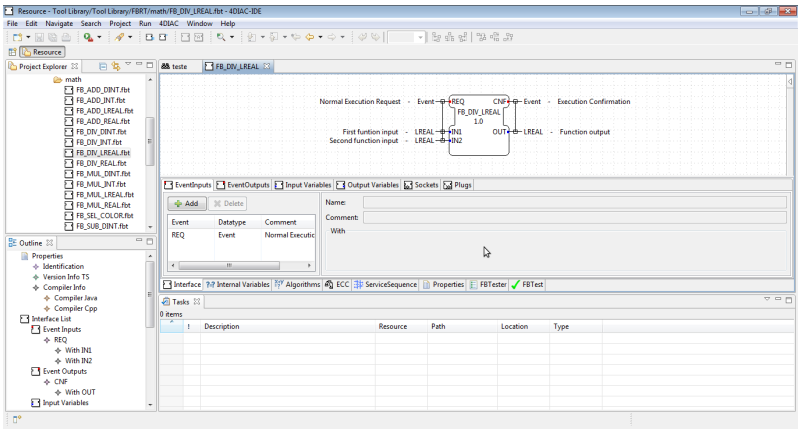


Figura 2.14: Janela do 4DIAC.

2.2.2 IDE's Comerciais

De forma a poderem ser usados sem limitações é necessário pagar pelos IDE's comerciais. Na sua maioria têm também a desvantagem de suportarem linguagens de programação não definidas pela norma IEC 61131-3, o que impossibilita a combinação de componentes de programas e a reutilização de software de alto nível, indo um pouco contra a espírito por detrás da criação da norma.

Testámos vários IDE's comerciais por forma a perceber melhor quais as características mais relevantes para o nosso IDE.

2.2.2.1 CoDeSys

O CoDeSys é desenvolvido e comercializado pela empresa de software alemã *3S-Smart Software Solutions*, desde 1994. (Ver Figura 2.15.) Suporta todas as linguagens IEC 61131-3, bem como uma linguagem gráfica própria, de nome *Continuous Function Chart*, muito próxima da FBD, mas oferecendo mais liberdade ao utilizador. Integra também suporte para a linguagem UML e permite integração de sistemas de controlo de versão [15].

2.2.2.2 ISaGRAF

O ISaGRAF é um IDE desenvolvido pela companhia *Rockwell Automation*, sediada nos Estados Unidos da América, e é baseado no Microsoft® Visual Studio®. Tal como o IDE 4DIAC,

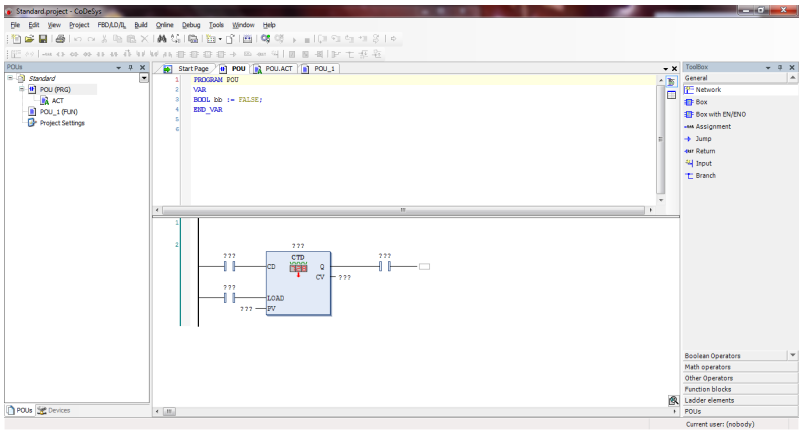


Figura 2.15: Janela do CoDeSys.

também suporta as linguagens da norma IEC 61131-3. Para além destas suporta também a construção de diagramas SAMA[2] (Figura 2.17) e, como o IDE CoDeSys, suporta sistemas de controlo de versão [16]. Pode ver o software na Figura 2.16.

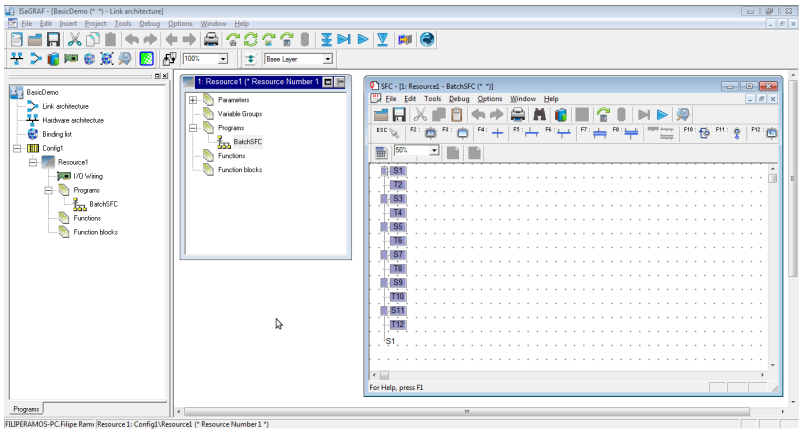


Figura 2.16: Janela do ISaGRAF.

[2]Diagramas usados na indústria da energia para implementação de estratégias de controlo. Foca-se no fluxo de informação dentro do sistema ao invés da interconetividade dos processos.

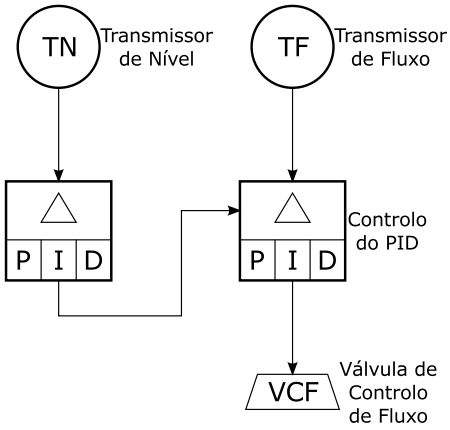


Figura 2.17: Diagrama SAMA [17, adaptada].

2.2.2.3 Unity Pro

O Unity Pro é desenvolvido pela companhia francesa *Schneider Electric* (ver Figura 2.18). Tal como os anteriores, suporta na íntegra as linguagens definidas pela norma IEC 61131-3, bem como a linguagem 984 LL (984 Ladder Logic) para PLC's Modicon [18].

Dos IDE's testados é aquele que apresenta uma curva de aprendizagem mais elevada, e o interface com o utilizador menos intuitivo.

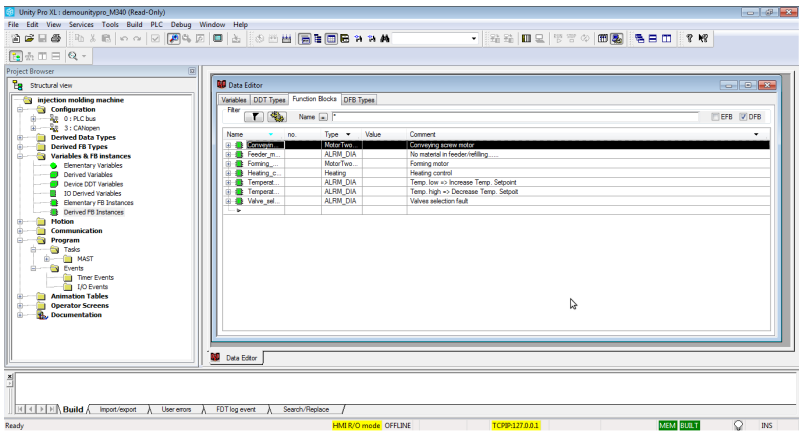


Figura 2.18: Janela do Unity Pro.

2.3 Eclipse

Vistos os principais IDE's existentes no mercado, vamos agora explorar o ambiente de desenvolvimento Eclipse, no qual se vai integrar o nosso IDE.

O Eclipse (ver Figura 2.19) é um IDE *Open Source* concebido primeiramente com o intuito do desenvolvimento de programas em Java. Foi criado pela IBM que o doou à comunidade de software livre. Rapidamente se tornou a primeira escolha de muitos utilizadores, e tem vindo a

crescer desde então. A sua estrutura é composta de forma modular o que o torna altamente configurável, permitindo a criação de novos módulos programados em Java que comunicam entre si através de *extension points* (Secção 2.3.4). Como base de todos os módulos corre uma plataforma de execução [19].

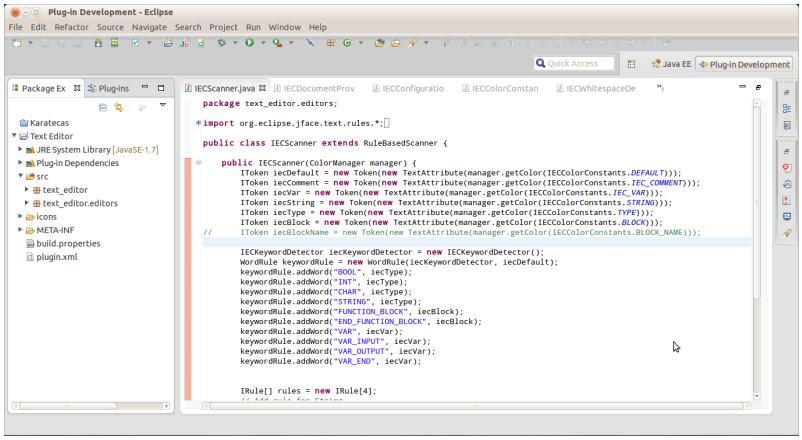


Figura 2.19: Janela do Eclipse.

Os módulos que compõem o Eclipse estão divididos em subsistemas (ver Figura 2.20), dos quais os mais relevantes são o *workbench* (Secção 2.3.1) e o *workspace*. O primeiro é composto pela janela principal do programa e as janelas e menus que estão no seu interior, o último gere os dados dos projetos com os quais se está a trabalhar e as preferências do utilizador — pode ser visto como um gestor do ambiente de trabalho. É possível o utilizador ter vários *workspaces* como forma de melhor organizar o seu trabalho, *e.g.*, tendo diferentes tipos de projetos em diferentes *workspaces*, pois a cada *workspace* está associado um diretório em disco, onde ficam guardados os projetos e os ficheiros com as preferências. É possível importar um projeto para um *workspace*, e nesse caso apenas o *link* para o projeto é guardado no diretório. Só um *workspace* pode estar ativo num dado *workbench*, num dado espaço de tempo.

Os módulos podem também ser divididos em duas categorias: *views* e *editors*. Ambos aparecem como janelas no *workbench*. As suas descrições são feitas em pormenor nas Secções 2.3.2 e 2.3.3.

Através da utilização de módulos, o Eclipse pode usar-se não só para programar em Java, mas também em muitas outras linguagens de programação. Basta para isso que se desenvolvam os módulos necessários. O nosso projeto criará um conjunto de módulos que possibilitem a programação das linguagens IEC 61131-3.

O Eclipse contém também uma *framework* que facilita a criação de elementos gráficos — a *framework* GEF, bem como outra que facilita a criação de metamodelos (definições das classes, regras e restrições a serem impostas na criação de um determinado modelo) — a *framework* EMF. Ambas foram importantes no desenvolvimento dos editores gráficos, e são descritas nas Secções 2.3.6 e 2.3.7.

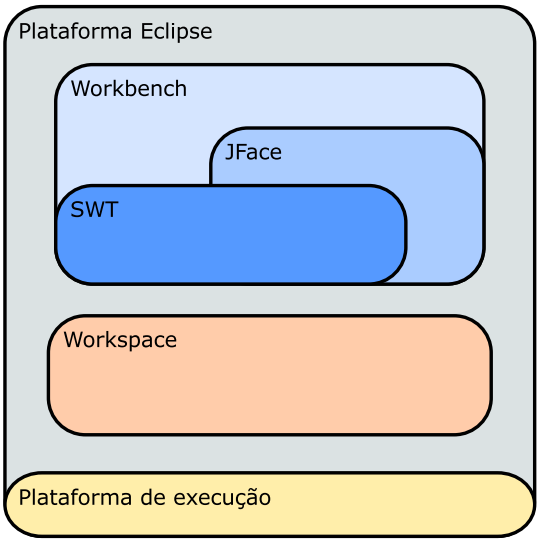


Figura 2.20: Arquitetura do Eclipse [20].

2.3.1 Workbench

Ao iniciar o IDE Eclipse apresenta-se-nos o *workbench*, ou bancada de trabalho. É no *workbench* que temos à nossa disposição os módulos do Eclipse que estão ativados, cada um numa janela própria. Podemos dispor esses módulos de várias formas para criar uma *perspective*. Cada *perspective* define a forma como as janelas dos módulos estão dispostas no janela principal (*workbench*), bem como quais os módulos a apresentar. Cada utilizador pode criar e gravar várias *perspectives* para uso em diferentes contextos. Pode ver-se na Figura 2.21 a organização descrita.

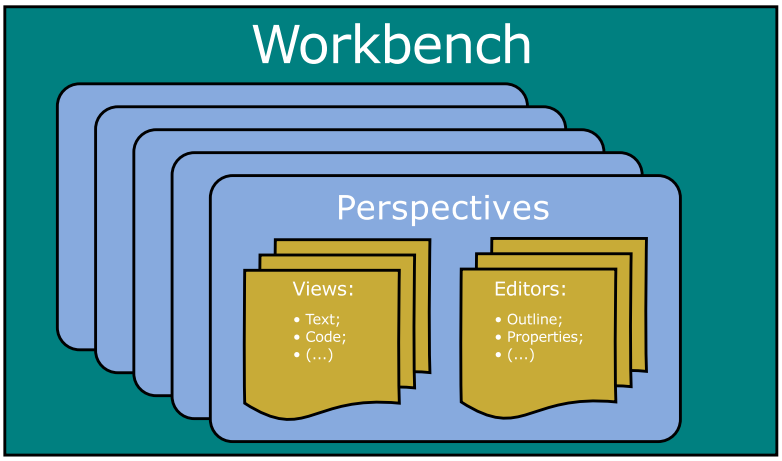


Figura 2.21: Organização do workbench Eclipse [21, adaptada].

O *workbench* assenta na ferramenta JFace e no *Standard Widget Toolkit* (SWT). O SWT é um sistema de *widgets* que permite o acesso às funcionalidades do interface com o utilizador do sistema operativo. O programador não tem de se preocupar com as diferenças entre os sistemas operativos onde o seu programa vai estar instalado, pois o SWT faz a "tradução" necessária. Os

programas tornam-se assim mais facilmente portáveis, e com diferenças mínimas no interface com o utilizador.

A ferramenta JFace fornece classes que ajudam o desenvolvimento de blocos de interface com o utilizador. Trabalha por cima de um sistema de *widgets* (neste caso, do SWT), ajudando em tarefas como ordenação de itens, especificação de ações do utilizador e efetuando a gestão dessas mesmas ações, fornece caixas de diálogo padrão, *wizards*, *etc.* Permite assim que o programador se foque no desenvolvimento dos seus programas sem perder demasiado tempo com tarefas comuns a qualquer aplicação que trabalhe com interfaces com o utilizador.

2.3.2 Views

As *views* são módulos cuja principal tarefa é mostrar informação (*e.g.*, uma janela onde aparecem listados todos os ficheiros de uma pasta). Serve também para mostrar propriedades de recursos abertos no sistema, como por exemplo as propriedades de determinado POU ou de determinado projeto. Em alguns casos é possível abrir um *editor* ao fazer duplo clique num recurso visível numa *view* (*e.g.*, se a *view* mostrar ficheiros Java, ao fazer duplo clique sobre eles o Eclipse abre-os automaticamente no seu editor Java).

2.3.3 Editors

Os *editors* são módulos que possibilitam a edição de ficheiros e outros recursos do sistema (*e.g.*, um editor textual). Podem estar associados a tipos específicos de ficheiros, de modo que ao selecionar um ficheiro, o Eclipse sabe exatamente qual o *editor* a usar. A informação a apresentar por um *editor* pode estar em forma de texto ou em forma de diagramas.

Os principais módulos no nosso projeto vão ser *editors*: os editores textuais para ST e IL, e o editor gráfico para SFC.

2.3.4 Extension Points

Os *extension points* são pontos de comunicação entre módulos. São descritos e configurados em ficheiros XML. Se o programador criar um módulo onde defina um *extension point*, outros programadores podem depois criar módulos que comuniquem com esse mesmo módulo através do *extension point* criado. O Eclipse fornece alguns *extension points* de raiz, dos quais nós utilizamos, por exemplo, os *editors*, onde se ligam os módulos com o mesmo nome, e o *menus*, onde se ligam módulos que definem menus.

2.3.5 Padrão de Arquitetura MVC

O padrão de arquitetura Modelo-Vista-Controlador (Model-View-Controller — MVC) é usado em engenharia de *software* na programação de interfaces com o utilizador. Consiste na divisão de uma aplicação em três partes interligadas, por forma a separar o modo como a informação é internamente representada (modelo), do modo como essa informação é apresentada ao, ou recebida

do utilizador (vista). Esse processo é mediado por um controlador (ver Figura 2.22). O Eclipse usa este padrão em alguns dos seus módulos que trabalham com visualização de informação. No padrão MVC, como utilizado no Eclipse, a componente central é o modelo, que consiste na representação interna da informação e na lógica para a sua manipulação. A vista é a forma de mostrar essa informação ao utilizador. O controlador aceita *input* do utilizador e converte-o em comandos para a vista ou para o modelo, permitindo ao utilizador manipular os dados e a sua visualização. Podem existir diferentes vistas para mostrar a mesma informação (*e.g.*, uma vista com um gráfico e outra com uma tabela podem ter como base o mesmo modelo).

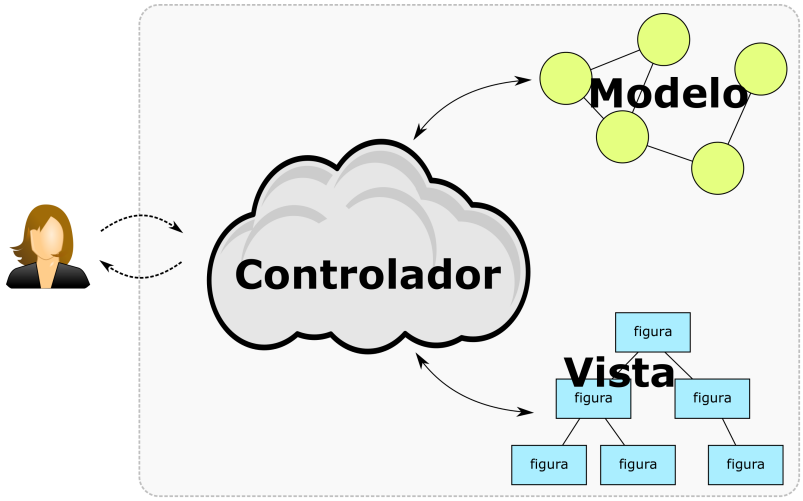


Figura 2.22: Padrão de Arquitetura MVC [22, adaptada].

2.3.6 Graphical Editing Framework

A *framework* de edição gráfica (Graphical Editing Framework - GEF) permite criar interfaces com o utilizador ricos e interativos, que não são facilmente criadas com recurso ao SWT. É composta por três *plug-ins*:

- Draw2d (*org.eclipse.draw2d*);
- GEF (*org.eclipse.gef*);
- Zest (*org.eclipse.zest*).

Dado que um dos *plug-ins* tem o mesmo nome da *framework*, sempre que nos quisermos referir à *framework* usaremos o nome GEF, e sempre que nos quisermos referir ao *plug-in* usaremos o nome GEF (MVC).

2.3.6.1 Draw2d

O *Draw2d* é um *toolkit* leve que permite desenhar sobre um *canvas* SWT. O *toolkit* define elementos de desenho aos quais dá o nome de figuras. Cada figura é um objeto Java que não utiliza

qualquer recurso do sistema operativo — daí ser um *toolkit* leve. As figuras podem conter figuras filhas incorporadas. As suas representações gráficas no *canvas* têm sempre limites retangulares.

Cada figura é associada a um *canvas* SWT, e o *Draw2d* coordena o desenho no *canvas* com as alterações na figura, e trata também dos eventos gerados pela interação do utilizador com o *canvas* SWT.

2.3.6.2 GEF (MVC)

O GEF (MVC) é uma *framework* interactiva construída com base no *toolkit* Draw2d, e que implementa o padrão de arquitetura MVC. Dado um qualquer modelo, o GEF permite gerar vistas utilizando figuras *Draw2d*. Faz todo o trabalho de um controlador, permitindo também a gestão da interação do rato e do teclado com as figuras.

A cada *canvas* SWT está associada uma classe que funciona como controlador, e a interliga com o respetivo elemento do modelo. A essa classe dá-se o nome de *EditPart*. São essas as classes responsáveis pelo desenho da figura (ver Figura 2.23).

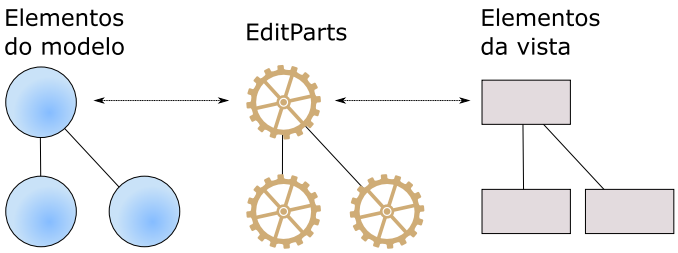


Figura 2.23: Funcionamento dos controladores EditPart [22, adaptada].

2.3.6.3 Zest

O *Zest* é um *toolkit* de visualização construído com base no *toolkit* Draw2d. Tem como objetivo principal facilitar a programação de diagramas baseados em grafos (ver Figura 2.24). No *Zest* cada grafo é um *widget* encapsulado numa figura JFace, permitindo ao utilizador programar diagramas de grafos como se estivessem a programar figuras JFace padrão.

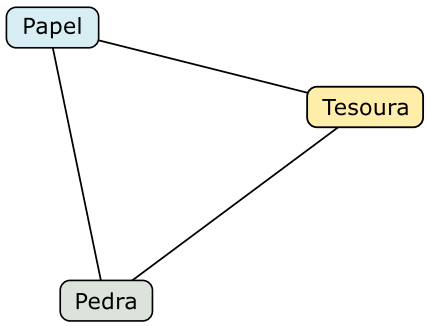


Figura 2.24: Tipo de grafo possível de criar com Zest [22, adaptada].

2.3.7 Eclipse Modeling Framework

A construção de editores gráficos é facilitada através da construção de modelos (de facto, quando se usa a *framework* Graphiti — ver Secção 2.3.8 — o uso de modelos é compulsivo). Os modelos são representações da estrutura de um diagrama: contêm informação sobre o tipo de elementos que fazem parte do diagrama, bem como da relação entre esses mesmos elementos. Pode ver um modelo simples para diagramas de Máquinas de Estados na Figura 2.25. Nele estão representados dois tipos de objetos: Estado e Transição. Cada Estado contém uma referência para uma Transição, e cada Transição uma referência para um Estado. Cada Estado contém também um atributo com a ação a executar quando este fica ativo, e cada Transição contém uma condição que permite a passagem para o próximo estado.

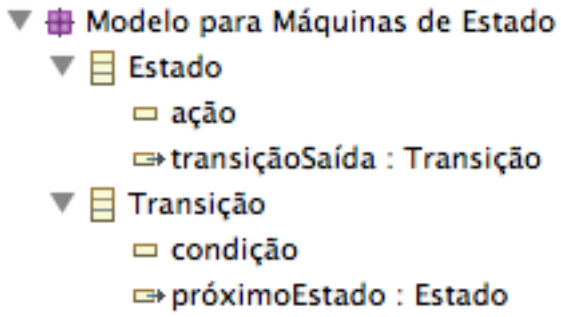


Figura 2.25: Modelo simples.

O modelo da Figura 2.25 foi construído usando a *framework* de modelagem do Eclipse (Eclipse Modeling Framework — EMF). Ela contém ferramentas que permitem especificar um modelo usando apenas definições das classes, dos seus atributos e relações. Os modelos podem ser definidos através de diagramas UML, num ficheiro XMI, ou através de anotações Java.

O Eclipse define um dialeto UML próprio, para o qual são convertidos todos os modelos importados: o dialeto *ecore*. Os modelos que usam este dialeto são chamados modelos *ecore*. O modelo da Figura 2.25 é deste tipo.

A terminologia EMF distingue a estrutura de um dado modelo (o metamodelo), da instância desse metamodelo (o modelo). No resto do documento vamos adotar essa distinção.

Depois de criado um metamodelo *ecore*, o EMF permite traduzi-lo para um metamodelo Java. Esse metamodelo é composto por um conjunto de classes, sendo criadas classes para representar todos os objetos do metamodelo, e classes auxiliares com métodos para criar esses objetos. O programador pode inserir novos métodos e instanciar novas variáveis nessas classes, e essas alterações serão mantidas nas próximas vezes que o metamodelo Java for criado a partir do metamodelo *ecore*.

A criação do metamodelo Java é feita em dois passos: primeiro usamos o metamodelo *ecore* para criar um outro tipo de metamodelo (*genmodel*) que, para além da informação presente no metamodelo *ecore*, contém informações adicionais sobre como deve ser gerado o metamodelo Java. De seguida o modelo *genmodel* é usado para gerar o metamodelo Java.

2.3.8 Graphiti

O Graphiti é uma *framework* que permite o desenvolvimento de editores de diagramas baseados em metamodelos, de uma forma mais estruturada e simples que o GEF. Os editores de diagramas criados com Graphiti já trazem por omissão uma barra lateral com os tipos de objetos gráficos que se podem criar, uma grelha para melhor organização dos elementos gráficos, *etc.* (ver Figura 3.9).

O Graphiti utiliza o GEF para criar diagramas, e trabalha com metamodelos gerados com o EMF. Para trabalhar com o Graphiti, o programador não necessita conhecer a *framework* GEF, apenas as *frameworks* Graphiti e EMF. O programador utiliza o EMF para criar um metamodelo, e depois trabalha com o Graphiti para criar elementos do modelo e os respetivos elementos gráficos.

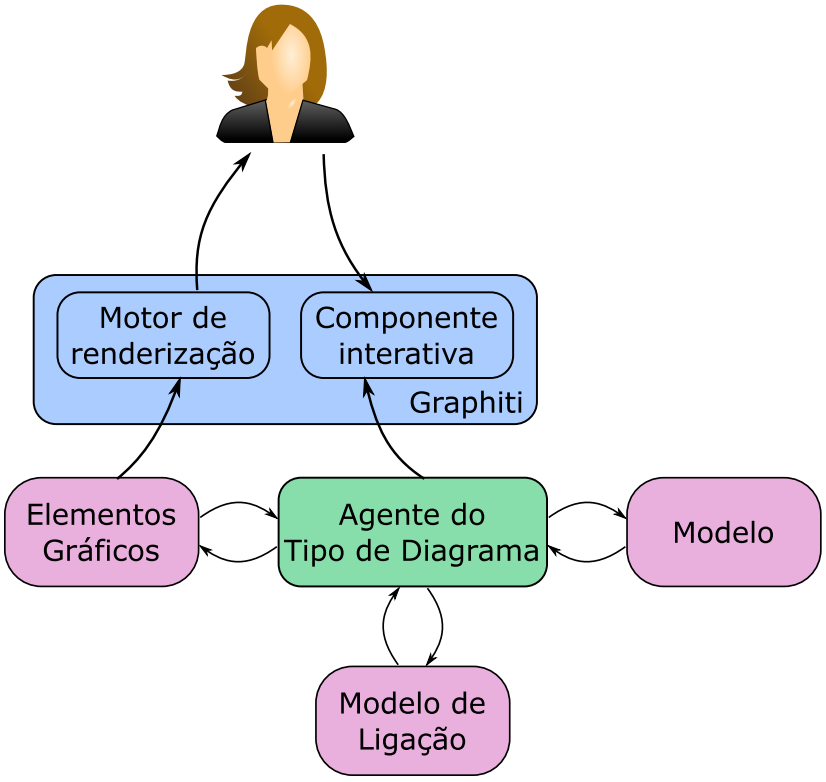


Figura 2.26: Estrutura do *framework* Graphiti [23, adaptada].

Os blocos principais da *framework* podem ser visto na Figura 2.26. O bloco "Elementos Gráficos" contém as representações gráficas dos elementos do modelo. O bloco "Modelo" contém o modelo gerado com base num metamodelo previamente criado. O bloco "Modelo de Ligação" contém a informação sobre quais elementos do modelo são representados por quais elementos gráficos, (um elemento gráfico pode representar mais do que um objeto do modelo, e por isso pode estar ligado a mais que um objeto do modelo). O bloco "Agente do Tipo de Diagrama" (Diagram Type Agent — DTA) é o único bloco que é necessário construir. Ele deve conter toda a lógica

para responder às ações do utilizador, criando e eliminando elementos gráficos e elementos do modelo (e respetivas ligações no Modelo de Ligação); movendo e alterando a forma dos elementos gráficos; *etc.* Pode ver uma representação da estrutura do DTA na Figura 2.27.

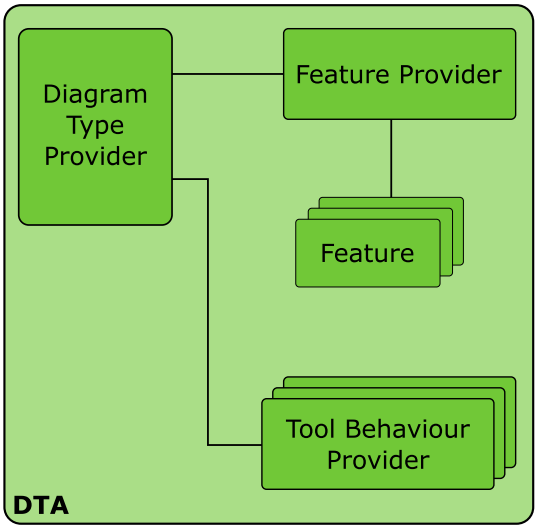


Figura 2.27: Estrutura do DTA [24, adaptada].

O DTA contém a classe *Diagram Type Provider* que define o diagrama que se está a desenvolver. Ela fornece um ou mais *Tool Behaviour Providers* que definem o comportamento do editor em diversas situações: *e.g.*, define o que apresentar na barra lateral, como devem ser geridos os duplos cliques ou a seleção de objetos no editor, que menus e botões de contexto devem ser criados, como é gerido o *zoom*, *etc.* Fornece também um *Feature Provider*, que define as funcionalidades do editor: chamadas de *features*.

A maior vantagem do Graphiti em relação ao GEF é a sua estrutura de mais alto nível, e uma API bastante simples e organizada. O desenvolvimento de um editor de diagramas em Graphiti foca-se na escrita das classes que implementam as *features*.

O Graphiti vem já com classes de base funcionais que implementam todas as *features* (as chamadas *Default Features*), que depois o programador pode estender, ou usar sem alteração: *e.g.*, o Graphiti já trás classes base que permitem apagar um objeto gráfico e respetivo objeto do modelo quando o utilizador prime a tecla *delete* tendo o objeto gráfico selecionado no editor. Se o programador não necessitar que mais nenhuma tarefa seja executada aquando da eliminação desses elementos, não tem necessidade de escrever nenhum código relativo à função de apagar elementos do diagrama.

As *features* implementam vários tipos de ações, e estão divididas por tipos, sendo as mais usadas as *Create Features*, para adicionar elementos ao modelo; as *Add Features*, para desenhar elementos gráficos no diagrama (representando elementos do modelo previamente criados); as *Delete Features* para apagar elementos do modelo; as *Remove Features*, para eliminar objetos gráficos do diagrama; e as *Move Features*, para mover objetos gráficos. Para além disso são

também necessárias *features* para criar as ligações entre os objetos gráficos (conexões): as *Add Connection Features*.

2.4 Compilador MATIEC

O MATIEC é um compilador *open source* para as linguagens da norma IEC 61131-3, suportando as linguagens para as quais existem representações textuais: *Instruction List*, *Structured Text* e *Sequential Function Chart*.

O desenvolvimento do compilador MATIEC (MatPLC's IEC compiler)[3] é coordenado pelo Prof. Dr. Mário Jorge Rodrigues de Sousa, é distribuído sob a licença GPL e é usado por diversas empresas, bem como pelo projeto Beremiz (ver Secção 2.2.1.1).

Do projeto MATIEC fazem parte dois tradutores de código: o *iec2iec* traduz código escrito numa das linguagens IEC 61131-3 para código escrito na mesma linguagem (serve primeiramente para *debug*), o *iec2c* traduz código IEC 61131-3 para código na linguagem de programação ANSI C (ver Figura 2.28).

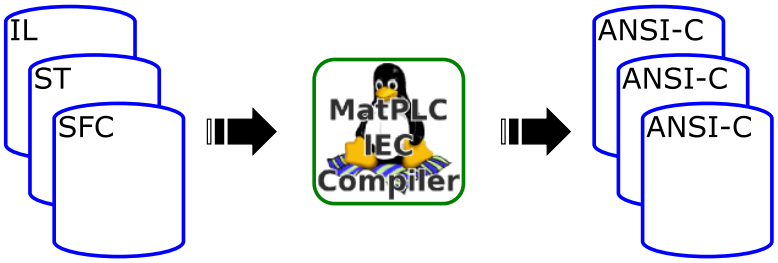


Figura 2.28: Função do compilador MATIEC [25, adaptada].

2.5 PLCopen XML

Em 1992 foi criada uma organização, independente do IEC, que se focou na criação de especificações e implementações em volta da norma IEC 61131, a fim de reduzir os custos em engenharia industrial — a PLCopen.

Com o intuito de criar um XML padrão para as linguagens IEC 61131-3, a PLCopen criou o grupo de trabalho *TC6 for XML*. Este definiu um interface, baseada em XML, que permite a transferência de toda a informação relevante de um projeto, criado por um certo *software*, para outro qualquer *software*. Esse XML padrão é o *PLCopen XML*, e encontra-se já na segunda edição (dezembro de 2008).

[3] O MatPLC é um softPLC que corre em linux (também sob a licença GPL) e que prima pela sua organização modular que permite a criação de módulos por terceiros e a sua integração conjunta num único interface de utilizador.

Capítulo 3

Desenvolvimento

Para o desenvolvimento de cada um dos editores do nosso IDE foram criados módulos independentes, mantendo assim uma programação mais modular e estruturada. (No Eclipse aos módulos programados para se ligarem ao IDE dá-se o nome de *plug-ins*.)

Cada *plug-in* integra-se no ambiente de desenvolvimento Eclipse através dos *extension points* que o ambiente tem para esse efeito, ou através de *extension points* criados por outros *plug-ins* (ver Secção 2.3.4).

Para melhor acompanhamento da descrição do desenvolvimento dos editores é recomendada a consulta dos diagramas de classes, que podem ser consultados no Anexo A. Ao longo do capítulo vão também aparecendo algumas parcelas desses diagramas, sempre que se achar necessário.

O desenvolvimento de *plug-ins* no Eclipse é feito com recurso ao *Plug-in Development Environment* (PDE), que fornece ferramentas para criar, desenvolver, testar, depurar e distribuir *plug-ins* Eclipse. Permite testar e fazer o *debug* dos *plug-ins* de uma forma fácil: quando queremos testar os nossos *plug-ins*, o PDE permite lançar um outro *workbench* Eclipse, idêntico àquele em que estamos a trabalhar, mas carregando os *plug-ins* que pretendemos testar. O novo *workbench* tem um *workspace* próprio e persistente entre execuções, o que permite realizar os testes sem termos de recriar os projetos de cada vez que queremos testar os *plug-ins*.

Um *plug-in* é descrito num manifesto XML (*plugin.xml*) cuja informação é lida pelo Eclipse por forma a proceder à sua ativação. Quando se inicia o ambiente de desenvolvimento Eclipse, ele lê os manifestos de todos os *plug-ins*, e quando forem necessários (*i.e.*, quando, pela ação do utilizador ou de outro *plug-in*, ele precisar de ser executado), efetua o seu carregamento — diz-se que é um carregamento do tipo *lazy loading*.

A gestão do ciclo de vida de um *plugin* é feita pelo Eclipse, mas pode ser alterada pelo programador recorrendo a uma classe que estenda a classe **AbstractUIPlugin**. Quando se cria um projeto para o desenvolvimento de um *plug-in*, o Eclipse providencia automaticamente uma classe que estende **AbstractUIPlugin** (a classe **Activator**). Essa classe faz parte do código fonte do *plugin*, e é o ponto de entrada do mesmo: é essa a classe que é chamada quando o *plug-in* precisa

de ser carregado. É também esta a classe que o programador deve editar caso necessite alterar o comportamento padrão da gestão do ciclo de vida do *plug-in*. No caso dos nossos editores tal não foi necessário.

Os *plug-ins* que definem os nossos editores (quer os textuais quer o gráfico), são do tipo *Editor* (ver Secção 2.3.3), e por isso devem ligar-se ao *extension point* **org.eclipse.ui.editors**. Essa extensão é definida nos respetivos manifestos XML.

O Eclipse fornece o interface **IEditorPart**, que deve ser implementado por qualquer classe que pretenda definir um novo editor. O recomendado é criar uma classe que estenda **EditorPart** (que já implementa o interface **IEditorPart**), ou uma sua subclasse. No nosso caso estendemos a classe **TextEditor**, que é uma subclasse de **EditorPart**, e já contém métodos que a tornam ótima para a criação de editores textuais. O nome dessa classe deve ser incluído na declaração do *extension point*, no manifesto XML. Para além disso deve também conter um identificador, e pode também definir qual a extensão dos ficheiros que deve ser associada ao editor, um ícone para representar graficamente esses mesmos ficheiros, e um nome para o editor. Pode ver de seguida a declaração de um *extension point* para um editor:

```
<extension
    point="org.eclipse.ui.editors">
    <editor
        name="Editor de teste"
        extensions="txt"
        icon="icons/editor.gif"
        class="editors.TestEditor"
        id="org.my.testEditor">
    </editor>
</extension>
```

3.1 Editores Textuais

Para a criação de editores de texto, o Eclipse usa o modelo de arquitetura de software Modelo-Vista-Controlador, já visto anteriormente. O modelo é representado pela classe **Document**, que gere o documento, e a vista é representada pela classe **StyledText**, que gere a visualização do documento. O controlador é responsável pela interligação do documento com a vista, e embora no Eclipse, no que diz respeito à programação de editores, não exista nenhuma classe que implemente o controlador, as classes **DocumentProvider** e **ViewerConfiguration** são as que mais se aproximam desta função: a classe **DocumentProvider** é responsável pelo controlo do documento, e a classe **ViewerConfiguration** pelo controlo da vista.

A classe **StyledText**, responsável pela visualização do documento, é um objeto do SWT. O programador não tem de trabalhar diretamente com esta classe, pois o Eclipse disponibiliza uma outra que a encapsula, e que fornece métodos de mais alto nível para a sua manipulação — a classe **TextViewer**. Embora esta classe seja já bastante completa, existe ainda uma outra classe,

subclasse de **TextViewer**, que lhe adiciona algumas funcionalidades mais em linha com editores textuais para linguagens de programação (permite, por exemplo, exibir os números de linha) — a classe **SourceViewer**. É esta a classe que usamos no nosso projeto.

Pode ver no diagrama da Figura 3.1 a estrutura do modelo MVC usada na programação dos editores textuais, com o nome das classes que vamos usar para cada bloco.

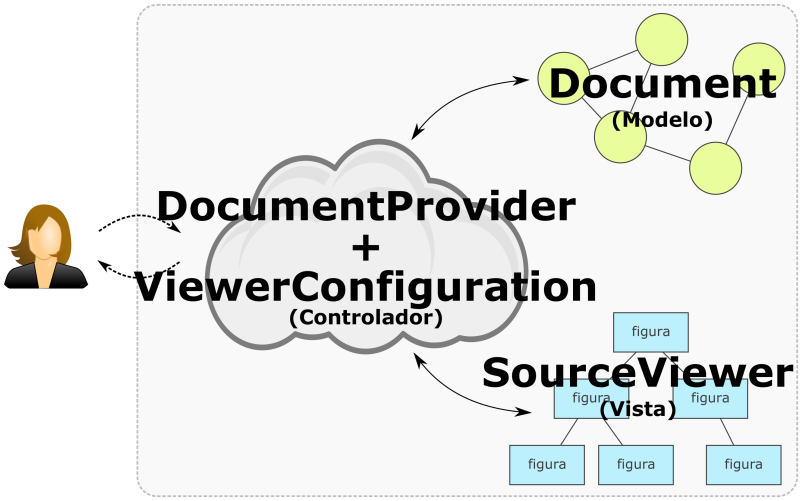


Figura 3.1: Modelo MVC nos Editores Textuais [22, adaptada].

O programador não trabalha diretamente com a classe responsável pela vista (**SourceViewer**). Ele deve configurá-la numa outra classe, que tem de estender **SourceViewerConfiguration**, e o Eclipse usa essa configuração na instanciação e configuração da classe **SourceViewer**. Para esse efeito foi criada a classe **ViewerConfiguration** (ver Secção 3.1.4 e Figura A.1).

Como criámos dois editores textuais, um para *Structured Text* e outro para *Instruction List*, criamos no manifesto XML duas extensões a **org.eclipse.ui.editors**. As classes que representam os editores são **STEditor** e **ILEditor** (ver Figuras A.1, 3.3 e 3.5), que como vimos atrás, estendem a classe **TextEditor**. Cada uma destas classes efetua a gestão de um documento e de uma vista para o editor em questão, sendo portanto responsáveis pela instanciação das classes que implementam o controlador (**DocumentProvider** e **ViewerConfiguration**) que por sua vez instanciam os objetos das classes **Document** e **SourceViewer**. Tanto a classe **DocumentProvider** como a classe **ViewerConfiguration** serão explicadas em detalhe nas Secções 3.1.3 e 3.1.4.

Um aspecto importante dos editores textuais é a sua capacidade de guardar os projetos num ficheiro em disco. No Eclipse a ligação entre um ficheiro em disco e um documento é feita através do interface **IDocumentProvider**, ou de uma classe que o implemente. No nosso caso optámos por estender a classe **FileDocumentProvider**, que implementa **IDocumentProvider**. Podemos assim assegurar a gravação de ficheiros em disco, e a abertura dos mesmos nos nossos editores. A classe criada chama-se **DocumentProvider** (ver Secção 3.1.3 e Figura A.1).

3.1.1 Modelo *Damage/Repair*

O modelo *Damage/Repair* serve para manter atualizados, de forma simples e otimizada, o documento e a sua representação visual. Quando um documento é alterado (*e.g.*, o utilizador escreveu ou apagou algo), ao invés de redesenhar todo o texto, o Eclipse descobre qual a partição (ver Secção 3.1.2) que foi alterada (calcula o dano feito ao texto) e redesenha-a (repara-a), sem perturbar as outras partições. Esse processamento é feito recorrendo a outras duas classes: um *Presentation Damager* e um *Presentation Repairer*: o *Presentation Damager* calcula a região do texto que deve ser reconstruída devido à alteração do documento, e a informação sobre qual a região do texto danificada é depois passada ao *Presentation Repairer* que a analisa, infere as alterações que são necessárias para a sua reparação, e repara-a.

Normalmente, e por uma questão de simplicidade, o *Presentation Damager* e o *Presentation Repairer* são representados apenas por uma classe à qual se dá o nome de *Presentation Damager Repairer*. As ações do *Presentation Damager Repairer* são coordenadas por uma outra classe, a que se dá o nome de *Presentation Reconciler*.

Pode ver uma descrição mais detalhadamente deste processo na Secção 3.1.4.2, onde se discute o funcionamento da *syntax highlight*.

3.1.2 Particionamento

No Eclipse, e no restante desta dissertação, à representação interna do conteúdo de um editor textual chama-se documento, e à representação visual do documento chama-se texto.

De cada vez que é detetada uma alteração no documento é executado um particionador (*Document Partitioner*), que serve para dividir o documento em zonas não sobrepostas — partições —, segundo regras definidas pelo programador. A cada uma dessas zonas é atribuído um identificador.

Um *Presentation Damager Repairer* pode ser associado a cada tipo de partição, permitindo assim que sejam tratadas de forma diferenciada. Nos nossos editores usámos essa diferenciação na aplicação da *syntax highlight* (ver Secção 1.4.1.1). A gestão dos *Presentation Damager Repairer* e das regiões a que estão associados é feita no *Presentation Reconciler*.

Pode ver uma descrição mais detalhadamente deste processo na Secção 3.1.4.2, aquando da discussão do funcionamento da *syntax highlight*.

3.1.3 Document Provider

Pode ver o diagrama de classes relacionadas com o *Document Provider* na Figura 3.3. Uma das funções da classe **DocumentProvider** é o particionamento do documento. Para isso ela necessita de um particionador, e este, por sua vez, necessita de um *scanner* — uma classe que varra o documento à procura de padrões. É através da configuração do *scanner* que o programador define a forma como o documento é particionado.

Como *scanner* para detetar partições o Eclipse fornece a classe **PartitionScanner**, que estende a classe **RuleBasedPartitionScanner**. Como queremos dividir o documento em dois tipos de

partição (comentário e código) que juntas formam todo o documento, apenas temos de detetar uma delas, e o restante pertence à outra partição. Optámos, por simplicidade, por detetar as partições que representam comentários.

Na classe **PartitionScanner** é definido um objeto do tipo **Token**[1] que serve de identificador da partição que contém comentários, e a regra a usar para a deteção dessa mesma partição. O Eclipse fornece classes que implementam regras de deteção de padrões nos documentos, e o seu funcionamento será melhor explicado quando se tratar da *syntax highlight* na Secção 3.1.4.2. Para a deteção de comentário usámos apenas uma dessas classes, a classe **MultiLineRule**. Esta classe permite detetar blocos de texto que comecem e terminem com determinadas sequências de carateres: *e.g.*, em ST um comentário é definido por uma sequência de carateres de início — `(*` — e uma sequência de carateres de fim — `*)`. O construtor desta classe recebe as sequências de carateres correspondendo, respetivamente, à sequência de início e fim do bloco a detetar, e o *token* a retornar caso o bloco seja detetado (neste caso, o identificador da partição).

O *scanner* é usado pelo particionador de modo a dividir o documento. O Eclipse fornece a classe **FastPartitioner** (que implementa **IDocumentPartitioner**) como um particionador simples e rápido, sendo apenas necessário que o programador lhe indique qual o *scanner* a usar. A sua configuração será melhor explicada na Secção 3.1.4.2. Pode ver-se um esquema da divisão de um documento nas respetivas partições na Figura 3.2.

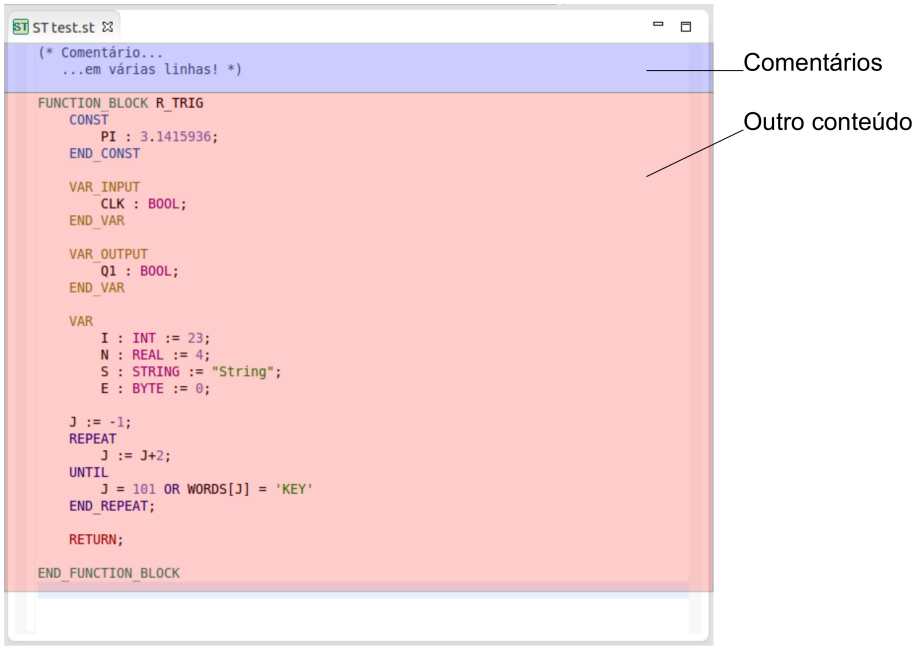


Figura 3.2: Particionamento do documento.

A classe **DocumentProvider** faz também a ponte entre o documento e os ficheiros em disco. É nesta classe que o documento é criado, como um objeto do tipo **Document**. É também esta

[1] A classe *Token* serve para encapsular objetos de qualquer tipo. Neste caso serve simplesmente para encapsular uma *String*, embora mais à frente seja usada para encapsular um objeto do tipo **TextAttribute**, que define atributos de texto, como a cor da face, ou a cor de fundo. (Daqui para a frente, qualquer objeto do tipo **Token** será referido como *token*.)

a classe responsável por guardar e abrir os ficheiros onde são guardados os documentos. Nesta classe fazemos o *override* do método *createDocument*, para criar um documento. De seguida criamos um objeto do tipo **PartitionScanner**, e fornecemo-lo ao construtor de um **FastPartitioner**. Usámos o método *void connect(IDocument document)* do **FastPartitioner** para que ele saiba qual o documento onde deve procurar as partições. Por outro lado, o documento também deve ser informado qual o particionador que lhe está associado. Para isso a classe **Document** fornece o método *void setDocumentPartitioner(IDocumentPartitioner partitioner)*. Pode ver o diagrama de classes relacionado com a classe *DocumentProvider* na Figura 3.3.

Cada uma das classes **STEditor** e **ILEditor** cria um objeto do tipo **DocumentProvider**, e para que esse objeto seja responsável pela criação e gestão dos respetivos documentos, ele é associado à respetiva classe com o método *void setDocumentProvider(FileDocumentProvider provider)*.

3.1.4 Viewer Configuration

A classe **ViewerConfiguration** é uma espécie de *hub* do editor. Aqui são inicializadas e configuradas várias das suas classes, incluindo a classe **SourceViewer**. Outros aspetos aqui configurados são a resposta ao duplo clique do rato, a *syntax highlight* e as classes responsáveis pela aplicação do modelo *Damage/Repair* às partições criadas pelo *DocumentProvider*.

Pode ver o diagrama de classes relacionadas com o *Viewer Configuration* na Figura 3.5.

3.1.4.1 Duplo clique

No Eclipse, a configuração por omissão do duplo clique permite que uma palavra seja selecionada quando um duplo clique é feito sobre ela, mas no nosso editor queremos também que quando houver um duplo clique no interior de um comentário, todo o comentário seja selecionado. Para isso criamos a classe **DoubleClickStrategy**, que implementa o interface **ITextDoubleClickStrategy**, e nela definimos métodos para determinar se o duplo clique se deu dentro de um comentário. Se for o caso, selecionamos todo o comentário, senão, apenas a palavra onde se deu o duplo clique.

3.1.4.2 Syntax Highlighting

A *syntax highlight* é executada sempre que seja detetado algum dano no documento, e por isso é executada a partir do *Presentation Reconciler*.

A coloração é feita de forma distinta em cada tipo de partição. A coloração dos blocos de comentário é feita no fim do particionamento do documento, e a coloração das *keywords* das linguagens é feita depois de uma segunda análise da partição de código, com o intuito de detetar as diferentes *keywords* das linguagens.

Depois de efetuada a partição do documento (ver Figura 3.2), a partição do código (aquela que é composta por tudo o que não é comentário) é de novo processada por um *scanner* que a divide em blocos de dois tipos: espaços em branco e texto. Para esse efeito o programador deve definir duas

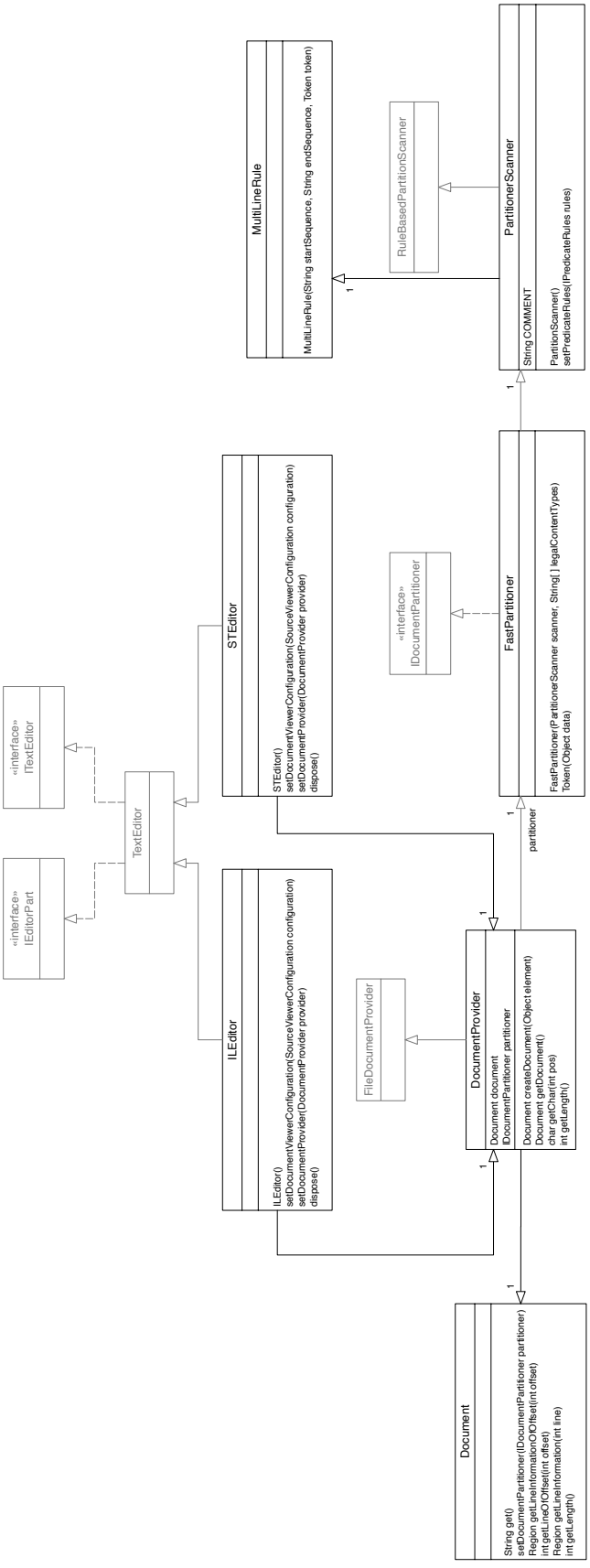


Figura 3.3: Diagrama de classes do *DocumentProvider*.

classes: uma que define o que é uma palavra; e outra, usada pela primeira, que define o que são espaços em branco. Os blocos de texto são encapsulado em objetos da classe **Token**, e são depois analisados, por ordem, à procura de padrões no seu interior, usando classes que implementam regras de pesquisa de padrões em documentos: *e.g.*, um *token* pode ser analisado por uma classe que implemente uma regra para detetar a palavra "FUNCTION", e se a palavra for detetada o *scanner* é avisado disso, o processamento desse *token* é terminado, e dá-se início ao processamento do próximo *token*. Caso o texto do *token* não seja a palavra "FUNCTION", o *token* é enviado à próxima classe detetora de padrões. A informação relativa aos padrões detetados é posteriormente usada pelo *scanner* para efetuar a coloração: a cada uma das regras detetoras de padrões usadas no *scanner* é associado um *token* com a informação de como devem ser colorido o texto caso o padrão seja detetado. Essa informação está presente num objeto do tipo **TextAttribute**.

O processo pode ser visualizado na Figura 3.4: primeiro o documento é dividido em dois tipos de partições (a vermelho e azul). Depois a primeira partição é dividida, por um *scanner*, em *tokens* (quadrados verdes) separados por espaços em branco. Finalmente esses *tokens* são analisados por classes que implementam regras de deteção de padrões, e o texto do documento representado pelos *tokens* é colorido de forma diferente, conforme o padrão detetado.

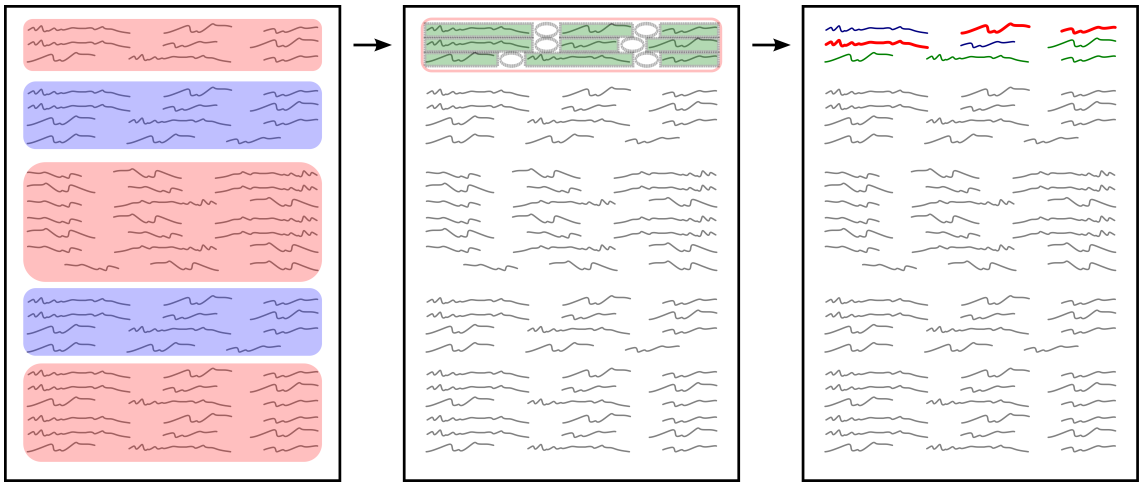


Figura 3.4: Particionamento e *scanning* do documento.

Quando o *Presentation Reconciler* associa um *Presentation Damager Repairer* a um tipo de partição, define também para essa associação qual o *scanner* a usar para efetuar a *syntax highlight*. Essa associação é realizada passando a classe que implementa o *scanner* ao construtor do *Presentation Damager Repairer*.

O *Presentation Damager Repairer* associado à partição de código é o *Presentation Damager Repairer* por omissão do Eclipse: a classe **DefaultDamagerRepairer**. O *scanner* que lhe é associado é representado pela classe **KeywordScanner**, que estende a classe **RuleBasedScanner**. Optámos por estender esta classe pois já contém métodos que facilitam o uso de regras para deteção de padrões. Dentro da classe **RuleBasedScanner** são definidos os atributos de texto a serem aplicados a cada tipo de *keyword*, e são instanciadas as regras a usar para a deteção das mesmas.

Para colorir os comentários, como são eles próprios uma partição, não é necessário o uso de nenhum *scanner*. Criámos por isso uma classe mais simples (a classe **PartitionDamager-Repairer**), que implementa os interfaces *IPresentationDamager* e *IPresentationRepairer*. Esta classe contém métodos que calculam a extensão da região que sofreu dano e de que forma essa região deve ser reparada. Depois de reparada a partição é colorida.

O Eclipse contém no pacote *org.eclipse.jface.text.rules* uma coleção de classes para deteção de padrões em textos, baseadas em regras (já atrás vimos uma dessas classes, quando tratamos da partição do documento: a classe **MultiLineRule** — ver Secção 3.1.2). O pacote fornece também *scanners*, dos quais vimos já a classe **RuleBasedPartitionScanner** — ver Secção 3.1.3.

Os principais *scanners* definidos no pacote são:

- **RuleBasedScanner** e
- **RuleBasedPartitionScanner**.

A classe **RuleBasedScanner** é a classe base dos *scanners*. As regras são-lhe comunicadas com o método *void setRules(IRule[] rules)*. Foi este o *scanner* usado para colorir o código dos editores.

A classe **RuleBasedPartitionScanner** é uma subclasse de **RuleBasedScanner**, e trabalha apenas com classes de regras que implementem **IPredicateRule**. Como o nome indica, serve primeiramente para criar partições no documento. Foi uma sua subclasse que usámos para particionar o documento nos nossos editores (ver Secção 3.1.2).

Todas as classes que definem regras recebem no seu construtor um ou mais *tokens* que devem ser retornados conforme os respetivos padrões sejam detetados ou não. Todas implementam o interface **IRule**. Foram algumas destas classes as usadas no *scanner* para aplicar a *syntax highlight* à partição de código. As principais classes de regras deste pacote são:

- **NumberRule**;
- **WordRule**;
- **WhitespaceRule**;
- **EndOfLineRule**;
- **SingleLineRule** e
- **MultiLineRule**.

A classe **NumberRule** serve para detetar números. O seu construtor recebe um *token* que deve ser retornado sempre que é detetado um número inteiro no texto. Não serve para detetar números em vírgula flutuante, nem em notação científica, pelo que não foi usada nos nossos editores.

A classe **WordRule** serve para detetar palavras. O seu construtor recebe um objeto de um tipo de classe que implemente **IWordDetector**, para que saiba como detetar palavras, *i.e.*, para que

saiba o que é uma palavra no contexto onde a classe vai ser aplicada. Recebe também um *token* para ser retornado sempre que uma palavra é detetada (*default token*). À classe **WordRule** podem ser adicionadas *Strings* associadas a *tokens* com o método *void addWord(String word, IToken token)*. Quando isto acontece, antes de a classe retornar o *default token* ao detetar uma palavra, verifica se essa palavra não é uma das adicionadas com o método *addWord*. Se for, retorna o *token* que lhe está associado ao invés do *default token*. Esta regra foi usada para detetar a grande maioria das *keywords* das linguagens. A classe criada para detetar palavras foi **WordDetector**. Nela se define qua uma palavra deve começar com uma letra, e deve conter letras, números ou o caratere '_' (*underscore*). Para além disso usámos também esta regra para detetar números. Neste caso a classe criada foi **NumberDetector**, e nela se define uma palavra como sendo um número, conforme usado nas linguagens ST e IL, incluindo em notação científica (*e.g.*, 2_133.32*E*5).

A classe **WhitespaceRule** serve para detetar espaços em branco. O seu construtor recebe um objeto de um tipo de classe que implemente **IWhitespaceDetector**, para que saiba o que se quer designar por espaços em branco, e sempre que um espaço em branco é detetado é retornado um *token* predefinido: *Token.WHITESPACE*. Nos nossos editores implementamos esta classe na classe **WhitespaceDetector**, que é instanciada no *scanner* usado na partição de código, por forma a dividir a partição em espaços em branco e outro tipo de texto (ver Figura 3.4).

As restantes classes diferem das anteriores pois apenas podem retornar um *token* (*success token*). Estas classes, para além de implementarem o interface **IRule**, também implementam o interface **IPredicateRule**.

A classe **EndOfLineRule** serve para detetar blocos de texto que comecem com um conjunto específico de carateres, e terminem no fim de uma linha. O seu construtor recebe uma *String* com o conjunto de carateres de início, e um *token* para ser retornado se um bloco assim definido for detetado. Esta classe não foi usada nos nossos editores.

A classe **SingleLineRule** serve para detetar blocos de texto que comecem e terminem num conjunto de carateres, não necessariamente iguais, e que não se propaguem por mais de uma linha. O seu construtor recebe duas *Strings*, que definem os carateres de início e de fim do bloco, e um *token* para retornar se o bloco for detetado. Esta regra foi usada para detetar *Strings* (começam e acabam com aspas duplas), e para detetar *Chars* (começam e acabam com aspas simples).

Por fim, a classe **MultiLineRule**, que foi já referida no particionamento do documento, serve para detetar blocos de texto definidos por uma sequência de carateres de início e de fim, tal como a classe **SingleLineRule**, mas que podem propagar-se por mais do que uma linha — daí ter sido usado para a deteção de comentários. O seu construtor recebe duas *Strings*, uma que define a sequência de início do bloco, e outra a sequência de fim. Recebe também o *token* que deve retornar caso o bloco seja detetado. Foi também usada para detetar blocos *Pragma*.

A classe **ViewerConfiguration** instancia uma classe auxiliar: a classe **ColorManager**. Esta classe faz a gestão das cores usadas no plugin. Disponibiliza o método *Color getColor(RGB rgb)* que transforma uma cor definida na classe **RGB** para a classe **Color**. A classe **RGB** guarda simplesmente os valores das componentes vermelho, verde e azul, ao passo que a classe **Color** usa

informações sobre o monitor, fornecidas pelo sistema operativo, para determinar os valores rgb que mais se adequam ao dispositivo. Por esse motivo os valores rgb fornecidos pela classe **Color** podem ser diferentes daqueles da classe **RGB** usada na sua criação.

Para facilitar a coloração do texto dos editores foi criado o interface **IColorConstants**. Nele estão guardadas as cores usadas no *plug-in* como objetos da classe **RGB**. Usamos a classe **RGB** por ser mais simples e facilitar uma futura integração com uma extensão que permita ao utilizador configurar as cores que prefere usar nos editores. Este interface é usado pela classe **ColorManager**.

A classe **ViewerConfiguration** disponibiliza métodos que são chamados para devolver classes de configuração específica (*e.g.*, contém um método que devolve a classe responsável pela configuração do duplo clique do rato). Esses métodos são:

- getConfiguredContentTypes();
- getDoubleClickStrategy() e
- getPresentationReconciler().

No método *getConfiguredContentTypes()* configurámos e retornamos um *array* com os *tokens* que definem os tipos de partições usadas pelo particionador: no nosso caso apenas retorna o *token* identificador dos comentários.

No método *getDoubleClickStrategy()* instanciámos e retornámos um objeto do tipo **DoubleClickStrategy**, a classe que gere o funcionamento do duplo clique.

Por fim, no método *getPresentationReconciler()* configurámos e retornámos um objeto do tipo **PresentationReconciler**. Configurámos também os *Presentation Damage* e *Presentation Repairer* para cada partição, com os métodos *void setDamager(IPresentationDamager damager, String contentType)* e *void setRepairer(IPresentationRepairer repairer, String contentType)*, e o *scanner* para usar na partição que contém o código.

Na classe **PartitionDamagerRepairer**, que faz a coloração dos comentários, tivemos de criar os seguintes métodos:

- endOfLineOf();
- getDamageRegion();
- addRange() e
- createPresentation().

No método *endOfLineOf()* determinámos e retornámos a posição do fim da linha onde se deu a alteração do documento.

No método *getDamageRegion()* calculámos e retornámos a extensão do documento que precisa ser reparada (que no máximo é igual à região da partição, mas pode ser menor se a partição contiver várias linhas, e apenas uma precise ser reparada).

No método *addRange()* atribuímos a cor definida para comentários a uma região de texto. Usámos este método no método *createPresentation()* para colorir a região alterada.

Cada uma das classes **STEditor** e **ILEditor** cria um objeto do tipo **ViewerConfiguration**, e para que esse objeto seja responsável pela criação e gestão das respetivas vistas, ele é associado à respetiva classe com o método *void setSourceViewerConfiguration(SourceViewerConfiguration configuration)*. Pode ver o diagrama de classes relacionado com a classe *ViewerConfiguration* na Figura 3.5.

3.1.5 Conversão para XML

Foi desenvolvida a conversão de documentos criados nos editores textuais para o formato XML, como definido na norma *PLCopen XML*. O *plug-in* criado para esse efeito cria um botão na barra de ferramentas e uma entrada num menu da barra de menus (ver Figura 3.6). Para realizar a conversão criámos a classe **Save2XMLHandler**. Nesta classe lemos o texto escrito no editor textual (seja ele de ST ou IL), e guardamo-lo numa variável do tipo *String — editorText*. Como o nosso IDE não permite a criação de projetos completos, tivemos necessidade de criar um projeto fictício para que a conversão para XML ficasse completa. Para isso criámos o método *createDummyProject()*, que cria um *header* e um *footer* com as informações do projeto. O texto final a ser guardado no ficheiro XML é composto pelo *header*, seguido do texto do editor, e por fim seguido do *footer*.

Depois do texto criado, é pedido ao utilizador que escolha onde guardar o ficheiro XML através de uma janela de diálogo. Feita a escolha o ficheiro é criado no local indicado. Pode ver na Figura 3.7 um ficheiro XML, resultante da conversão de um ficheiro em ST, e onde se mostra o *header*, e o *footer*, para além da zona onde se encontra o código lido do editor.

3.2 Editores Gráficos

A programação do editor gráfico para a linguagem SFC foi executada com base na *framework* Graphiti. Como vimos atrás, o Graphiti mantém duas estruturas paralelas, uma com elementos do modelo, e outra com os elementos gráficos que lhes estão associados. Cada elemento do modelo pode estar associado a mais do que um elemento gráfico, e vice-versa. Cabe ao programador desenhar os elementos gráficos, criar os elementos do modelo (usando o metamodelo Java — ver Secção 2.3.7), e criar a associação entre eles. Na secção seguinte explica-se o desenvolvimento do metamodelo *ecore* que está na base do metamodelo Java usado pelo Graphiti.

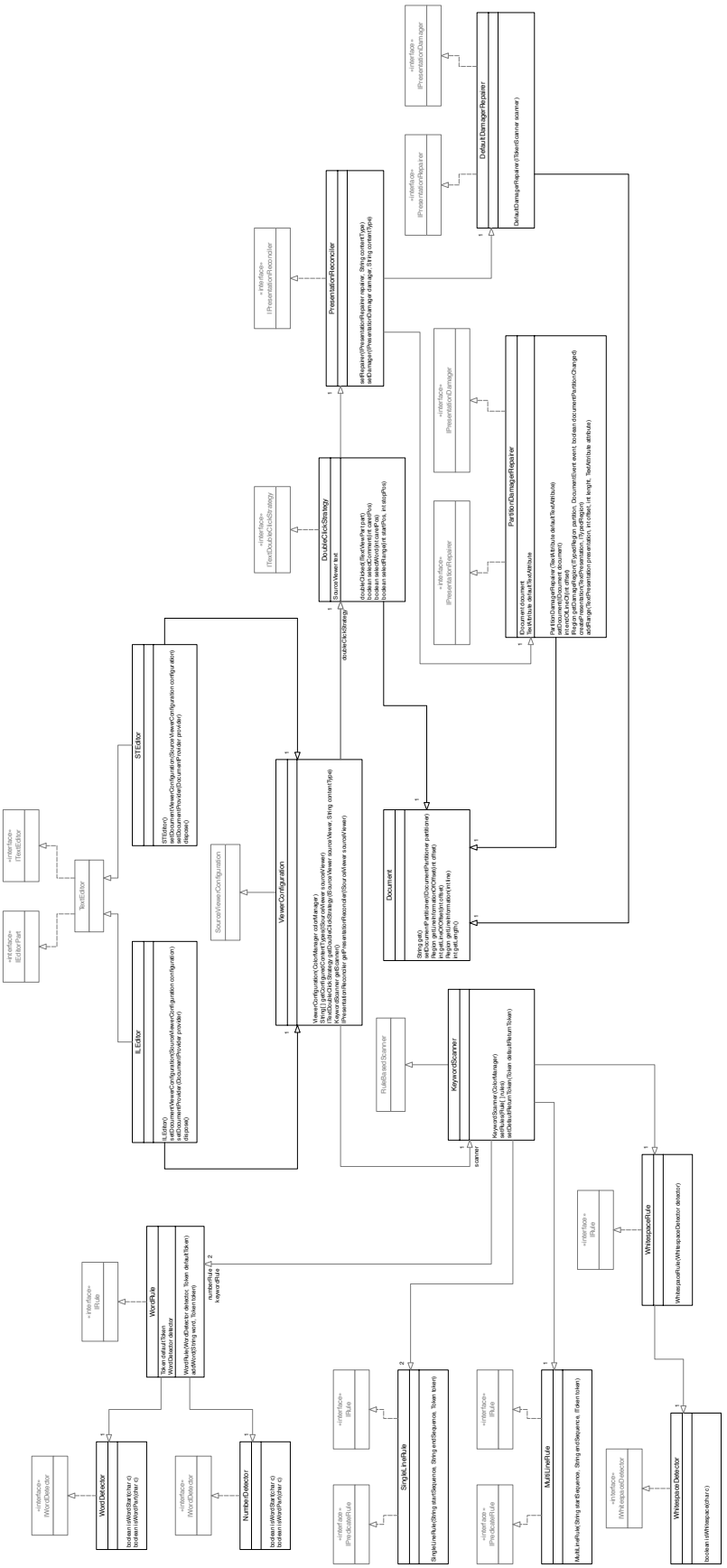


Figura 3.5: Diagrama de classes do *ViewerConfiguration*.

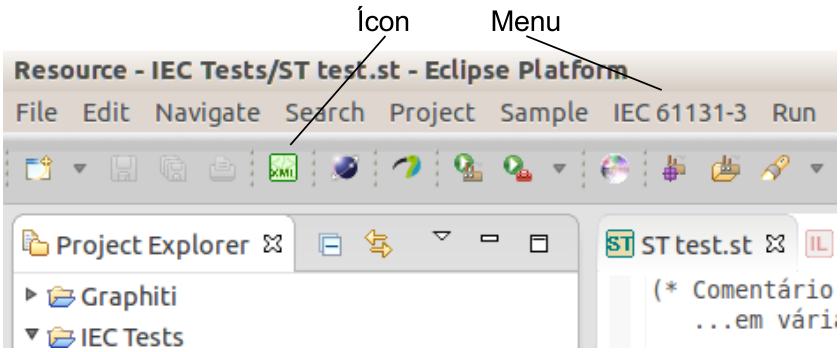


Figura 3.6: Ícon e Menu para conversão XML.

3.2.1 Metamodelo

Foi criado um metamodelo *ecore* (ver Figura 3.8), que serviu para o desenvolvimento do editor gráfico para a linguagem SFC. Nele foi criado um objeto do tipo *SFCDiagram*, que serve de base a todos os outros elementos. Funciona como a tela onde todos os outros elementos são desenhados. Existe também um objeto do tipo *ProjectData* que contém informação sobre o projeto: *e.g.*, tem um atributo do tipo *String*[2] onde se guarda o nome do projeto: *projectName*. Foi também criado um objecto do tipo *SFCObject*, que funciona como super tipo de todos os outros objectos que representam elementos gráficos da linguagem, contendo tributos que são comuns a todos, como o seu *id*, e o tamanho do objeto gráfico (comprimento e largura). O *id* do objeto é guardado no campo *localId*.

Foram também criados alguns objetos auxiliares, como o objeto *Position*, que guarda dois inteiros que representam as variáveis *x* e *y* da posição do elemento no diagrama, e o objeto *Connection*, que guarda vários objetos *Position*, que juntos representam os nós de uma conexão.

Foram também criados objetos para representarem os objetos gráficos da linguagem: foi criado o objeto *ActionBlock* que guarda as ações a serem executadas por um *Step*. Foi criado um objeto do tipo *Step* que contém um atributo para guardar o nome (*name*, do tipo *String*), e duas referências, uma para o objeto que o antecede e outra para o objeto que o sucede (*inObject* e *outObject*, ambas do tipo *SFCObject*). Contém também os booleanos *initialStep* e *macroStep* que permite definir se o *Step* em questão é do tipo *Initial Step* ou *Macro Step*. Contém também uma referência para um *actionBlock*. Foi criado um objeto do tipo *Transition* que contém duas referências: uma para o *SCFObject* anterior (*inObject*) e outra para o *SFCObject* posterior (*outObject*). Contém também a condição associada à *Transition*. Foram também criados objetos do tipo *SimultaneousDivergence*, *SimultaneousConvergence*, *SelectionDivergence* e *SelectionConvergence*, para representarem os objetos *SFC* que permitem criar e unir ramos paralelos. Estes objetos contêm vetores para guardar as referências dos objetos *Transition* e *Step* que lhes estão ligados. Por fim, foi criado o objeto *JumpStep*, que representa um *Jump Step*. Contém uma referência para o *SFCObject* ao qual está ligado, e uma referência para o *SFCObject* para onde a execução do programa deve saltar.

[2]Os metamodelos *ecore* utilizam os tipos de dados existentes em Java, mas para denotar que são tipos de um metamodelo, o seu nome começa com um 'E': *e.g.*, *String* passa a *EString*.

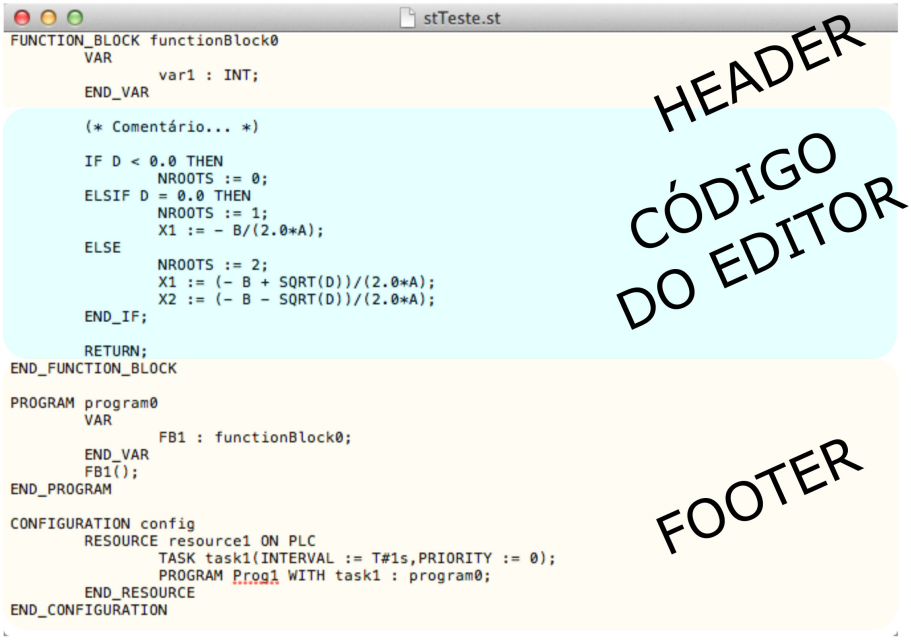


Figura 3.7: Ficheiro ST convertido em XML.

O objeto *SFCDiagram* contém referências para todos os objetos do metamodelo: o vetor *steps* recebe as referências das instâncias dos objetos de tipo *Step*, o vetor *transitions* recebe as referências das instâncias dos objetos de tipo *Transition*, *etc.*

3.2.2 Graphiti

O IDE Eclipse permite a criação destes editores recorrendo ao EMF e ao GEF. No entanto, a maioria da informação encontrada, quer na internet, quer em livros, tratam mais dos conceitos teóricos do GEF do que como o aplicar na prática. Por isso, e dada a elevada complexidade do GEF, foi escolhido desenvolver os editores usando uma ferramenta de mais alto nível, que trabalhasse por cima do GEF. Depois de alguma pesquisa a escolha recaiu sobre a *framework* Graphiti.

Para o desenvolvimento de editores gráficos usando Graphiti é necessário criar, no DTA, as classes que implementam as *features* que queremos implementar (ver Secção 2.3.8).

O metamodelo *ecore* descrito atrás foi convertido num metamodelo Java (ver Secção 2.3.8), de modo a poder ser utilizado pelo Graphiti. No metamodelo Java está presente uma classe que serve para criar todos os objetos definidos no metamodelo *ecore*: a classe **SequentialFunctionChartFactory**. Pode ver a janela do editor gráfico para a linguagem SFC na Figura 3.9, e o diagrama das classes relacionadas com o classe **SequentialFunctionChartFactory** na Figura 3.10.

O Graphiti fornece ao IDE Eclipse dois *extension points* próprios para a ligação de editores gráficos criados com a sua API: o *extension point* **org.eclipse.graphiti.ui.diagramTypes** e o *extension point* **org.eclipse.graphiti.ui.diagramTypeProviders**. O primeiro serve apenas para ligar

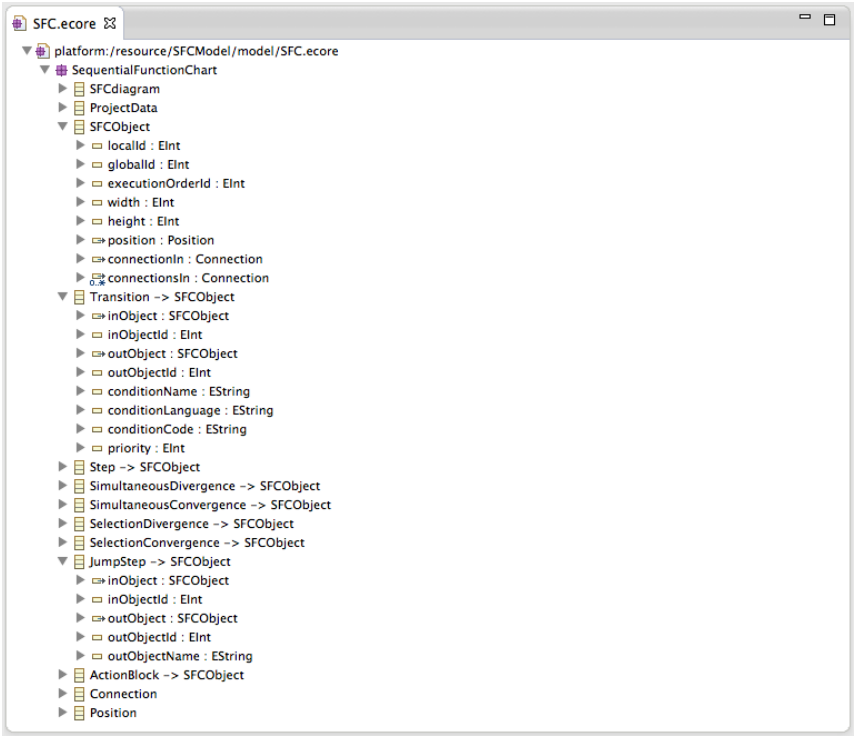


Figura 3.8: Metamodelo *ecore*.

uma definição de um tipo de diagrama: tem um nome, uma descrição e um *id*. Como serve apenas como uma definição, não está associado a nenhuma classe. O segundo serve para ligar o editor, especificando qual a classe que deve ser chamada para o inicializar, e qual o *id* do tipo de diagrama com que o editor vai trabalhar (definido no primeiro *extension point*). Tal como no caso dos editores textuais, o Eclipse cria a classe **Activator** para controlar o ciclo de vida do *plug-in*.

A classe que inicia o editor (como definido no *extension point*) deve implementar o interface **IDiagramTypeProvider** — no nosso caso criamos a classe **DiagramTypeProvider**, que como recomendado, estende a classe **AbstractDiagramTypeProvider** que já implementa o atrás mencionado interface. Nessa classe deve ser instanciado um objeto do tipo **AbstractFeatureProvider** que controla quais as funcionalidades (*features*) do nosso diagrama, para as quais o Graphiti não deve usar as *Default Features*. Para esse efeito criámos a classe **FeatureProvider**, que contém métodos para inicializar e retornar instâncias de todas as classes responsáveis por criar objetos gráficos, objetos do modelo, e criar as conexões. Inicializa também as classes que tratam da atualização dos objetos gráficos quando os respetivos objetos do modelo são alterados, e as classes para redesenhar as conexões quando os objetos gráficos são movidos. Essas classes são:

- *Create Features* (subclasses de *AbstractCreateFeature*):
  - CreateTransition;
  - CreateInitialStep;
  - CreateStep;

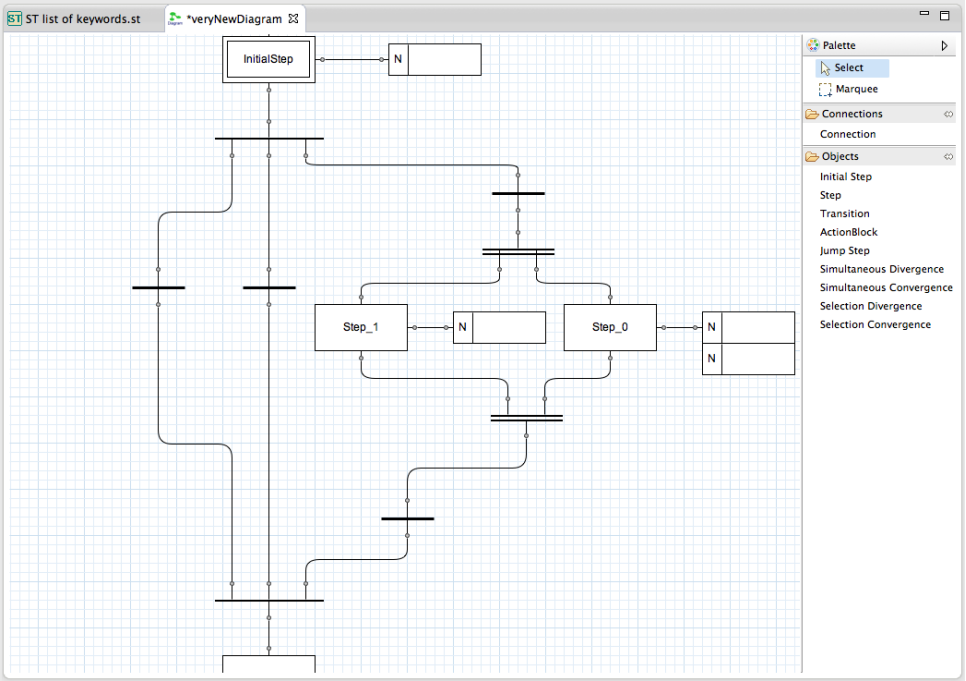


Figura 3.9: Editor gráfico para SFC.

  - CreateSimultaneousDivergence;
  - CreateSimultaneousConvergence;
  - CreateSelectionDivergence;
  - CreateSelectionConvergence;
  - CreateActionBlock;
  - CreateJumpStep.

- *Add Features* (subclasses de *AbstractAddFeature*):

  - AddTransition;
  - AddInitialStep;
  - AddStep;
  - AddSimultaneousDivergence;
  - AddSimultaneousConvergence;
  - AddSelectionDivergence;
  - AddSelectionConvergence;
  - AddJumpStep;
  - AddActionBlock;
  - AddConnection.

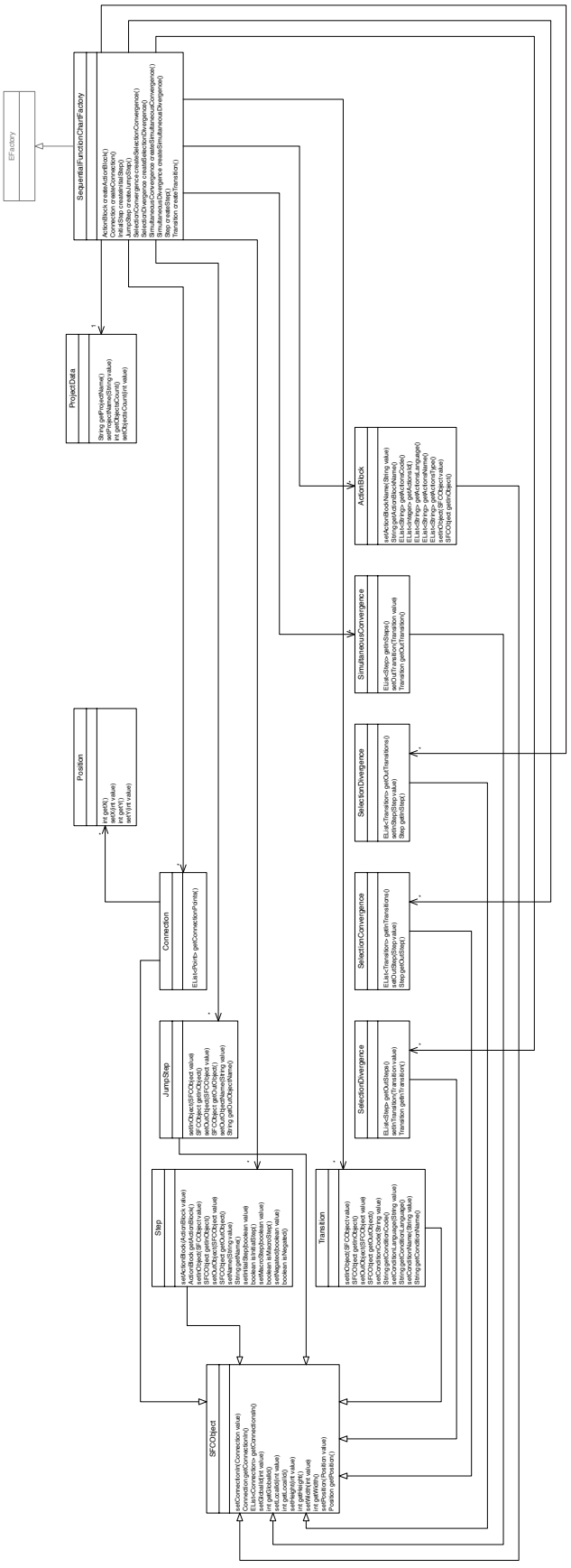


Figura 3.10: Diagrama de classes da **SequentialFunctionChartFactory**.

- *Create Connection Feature* (subclasse de *AbstractCreateConnectionFeature*):
  - CreateConnection.
- *Update Features* (subclasses de *AbstractUpdateFeature*):
  - UpdateSimultaneousConvergence;
  - UpdateSimultaneousDivergence;
  - UpdateSelectionConvergence;
  - UpdateSelectionDivergence;
  - UpdateActionBlock.
- *Move Element Feature* (subclasse de *DefaultMoveShapeFeature*):
  - MoveElement.
- *Move Bendpoint Feature* (subclasse de *DefaultMoveBendpointFeature*):
  - MoveBendpoint.
- *Add Bendpoint Feature* (subclasse de *DefaultAddBendpointFeature*):
  - AddBendpoint.
- *Resize Shape Feature* (subclasse de *DefaultResizeShapeFeature*):
  - ResizeElement.
- *Delete Feature* (subclasse de *DefaultDeleteFeature*:
  - DeleteWithoutPrompt.
- *Custom Features* (subclasses de *AbstractCustomFeature*):
  - IncreaseActions;
  - IncreaseConnections;
  - DecreaseConnections;
  - SetConnections.

Depois de criarmos as classes responsáveis pelas *features*, o Graphiti trata de criar a estrutura do editor, com uma barra lateral por omissão (ver Figura 3.9) onde podem ser escolhidos os elementos do modelo com representação gráfica, que se querem inserir no diagrama, e que permite também criar conexões. Quando o utilizador cria um elemento no diagrama, o Graphiti chama a respetiva classe para criar e desenhar o objeto gráfico (**Add***), e de seguida a classe para criar o objeto no modelo (**Create***), onde é atualizado o modelo de ligação. No caso das conexões o processo é invertido, e são primeiro criadas as referências no modelo (**CreateConnection**) e só depois criada a representação gráfica e atualizado o modelo de ligação (**AddConnection**).

3.2.2.1 Feature Provider

Pode ver o diagrama de classes relacionadas com o *Feature Provider* na Figura 3.11. A classe **FeatureProvider**, que gere as *features* do diagrama, retorna as instâncias das classes responsáveis pelas *Create Features* com o método *ICreateFeature[] getCreateFeatures()*, e a instância da classe responsável pela *Create Connection Feature* com o método *ICreateConnectionFeature[] getCreateConnectionFeatures()*. Estas são as classes responsáveis por criar os elementos no modelo.

As instâncias das classes responsáveis pelas *Add Features* são retornadas com o método *IAddFeature getAddFeature(IAddContext context)*[3]. Estas classes são responsáveis por criarem os elementos gráficos ligados aos elementos do modelo.

As instâncias das classes responsáveis pelas *Update Features* são retornadas com o método *IUpdateFeature getUpdateFeature(IUpdateContext context)*[4]. Estas classes servem para atualizar os elementos gráficos, sempre que os respetivos objetos do modelo tenham sofrido alterações.

A instância da classe responsável pela *Resize Shape Feature* é retornada com o método *IResizeShapeFeature getResizeShapeFeature(IResizeShapeContext context)*[5]. Esta classe serve para redimensionar os objetos gráficos.

A instância da classe responsável pela *Move Shape Feature* é retornada com o método *IMoveShapeFeature getMoveShapeFeature(IMoveShapeContext context)*[6]. Esta classe é responsável pelas alterações que necessitem de ser feitas ao diagrama quando um dos elementos gráficos é movido.

A instância da classe responsável pela *Move Bendpoint Feature* é retornada com o método *IMoveBendpointFeature getMoveBendpointFeature(IMoveBendpointContext context)*[7]. Esta classe trata das alterações a serem feitas a uma conexão quando um dos seus pontos de quebra (nós) é movido.

A instância da classe responsável pela *Add Bendpoint Feature* é retornada com o método *IAddBendpointFeature getAddBendpointFeature(IAddBendpointContext context)*[8]. Esta classe serve para inserir novos pontos de quebra a uma conexão.

A instância da classe responsável pela *Delete Feature* é retornada com o método *IDeleteFeature getDeleteFeature(IDeleteContext context)*[9]. Esta classe serve para apagar elementos do modelo. As *Delete Features* são oposta às *Create Features*. Por sua vez as *Add Features* têm as *Remove Features* como opostas, mas no nosso diagrama não necessitámos de implementar nenhuma *Remove Feature*.

[3] O interface **IAddContext** contém a informação do objeto que se quer desenhar.

[4] O interface **IUpdateContext** contém a informação sobre um objeto do modelo que sofreu alteração.

[5] O interface **IResizeShapeContext** contém a informação sobre um objeto gráfico que se pretende redimensionar.

[6] O interface **IMoveShapeContext** contém a informação sobre um elemento gráfico que foi movido.

[7] O interface **IMoveBendpointContext** contém a informação sobre um nó de uma conexão que foi movido.

[8] O interface **IAddBendpointContext** contém a informação sobre uma conexão onde foi inserido um novo nó.

[9] O interface **IDeleteContext** contém a informação sobre um objeto do modelo cujo elemento gráfico a que estava ligado foi removido.

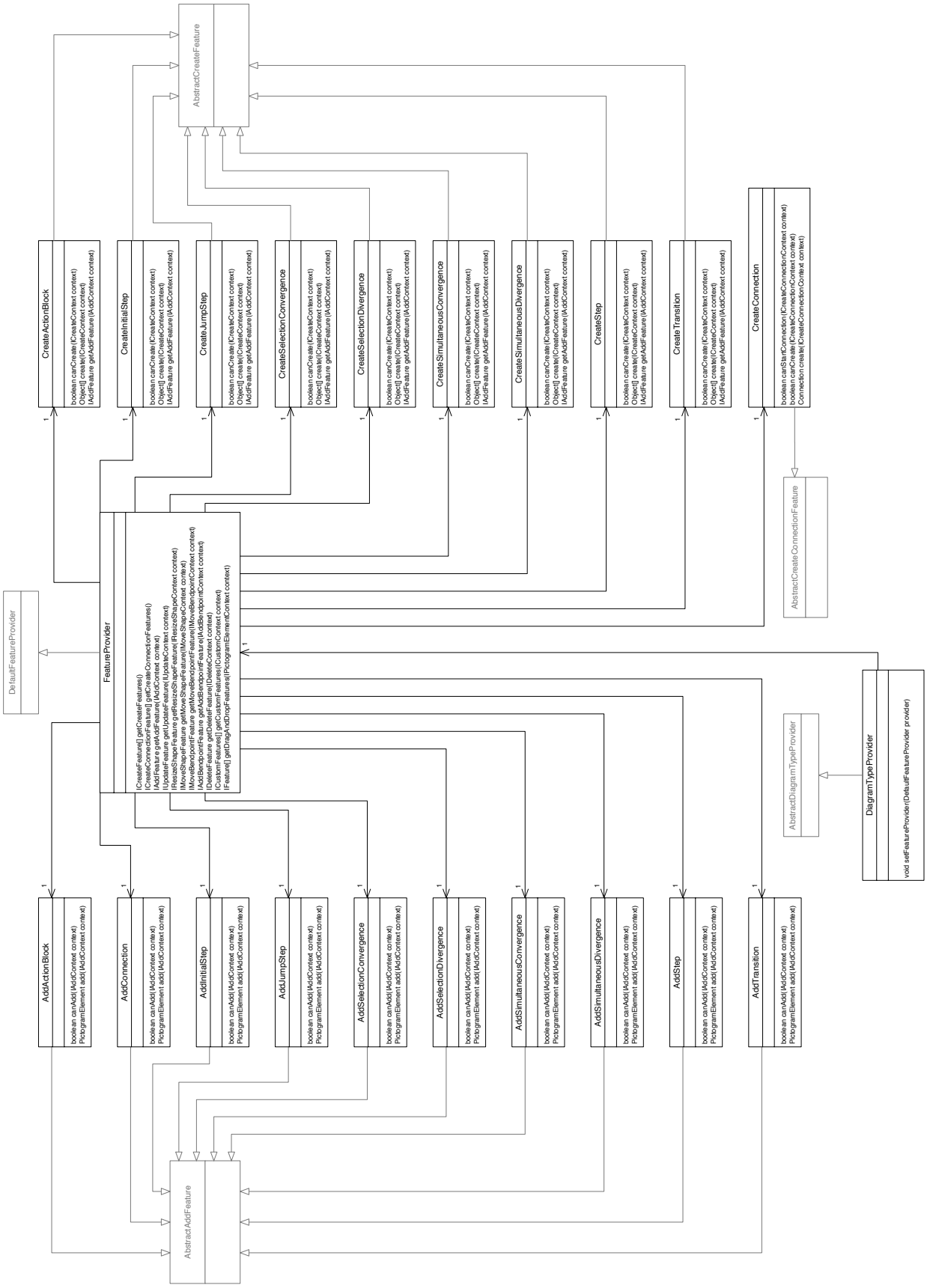


Figura 3.11: Diagrama de classes do *Feature Provider*.

As instâncias das classes responsáveis pelas *Custom Features* são retornadas com o método *ICustomFeature[] getCustomFeatures(ICustomContext context)*[10]. Estas são classes que fornecem *features* não definidas pelo Graphiti.

Para além destes métodos, o método *IFeature[] getDragAndDropFeatures(IPictogramElement-Context context)*[11] retorna as instâncias das classes que devem ser executadas quando um elemento gráfico é arrastado. No nosso caso é suficiente que este método retorne as mesmas instâncias do método *getCreateConnectionFeatures*.

3.2.2.2 *Add Features*

Cada uma das classes responsáveis pelas *Add Features* contém um método *canAdd()*, que verifica se o objeto que se quer inserir é do tipo correto (*e.g.*, do tipo *Transition* na classe **AddTransition**), e se o local onde se quer inserir o objeto é um diagrama. Retornam *true* em caso afirmativo, e *false* em caso negativo. Contém também um método *add()*, que usa os interfaces Graphiti para desenhar gráficos (*IGaService*) e pictogramas (*IPeService*), e cria um conjunto de objetos que depois de invocados geram as imagens que representam um objeto gráfico (ver Figura 3.12). Para se poderem criar conexões entre objetos gráficos, estes devem ter definidos pontos de ligação — âncoras. As âncoras são também definidas neste método, usando os mesmos interfaces — círculos cinzentos nos objetos da Figura 3.12. As imagens são depois desenhadas no diagrama quando, por último, se chama o método *layoutPictogramElement(PictogramElement)*[12].

As âncoras podem ser de três tipos: âncoras no limite da caixa; âncoras relativas à caixa; e âncoras de ponto fixo. As âncoras no limite da caixa localizam-se no centro do objeto. Contudo, a linha que representa a conexão não termina no centro do objeto, mas sim na interceção com o limite do objeto. As âncoras relativas à caixa situam-se num ponto cujas coordenadas são dadas de forma relativa, *e.g.*: a um quarto da largura e a um terço da altura. As âncoras de ponto fixo situam-se num ponto cujas coordenadas são dadas de forma absoluta, *e.g.*: 12 pixels a partir da esquerda e 23 pixels a partir do topo. Nos nossos objetos gráficos usámos âncoras relativas à caixa.

A *Add Feature* responsável por criar *Initial Steps* verifica se existe já algum *Initial Step* no diagrama. Se existir apresenta uma janela de diálogo com um erro a avisar o utilizador que um diagrama não pode conter mais do que um *Initial Step*. (ver Secção 3.2.3)

As *Add Features* responsáveis por criar *Steps* e *Initial Steps* apresentam ao utilizador uma janela de diálogo onde ele pode inserir o nome a dar ao *Step* ou *Initial Step*. Cada *Step*/*Initial Step* deve conter um nome, que deve ser único no diagrama. Se o utilizador inserir um nome inválido (com espaços, ou já repetido) é-lhe apresentada uma outra janela com o erro, e a pedir para inserir um novo nome (ver Secção 3.2.3).

[10] O interface **ICustomContext** contém a informação sobre um qualquer objeto.
[11] O interface **IPictogramElementContext** contém a informação sobre um objeto gráfico.
[12] A classe **PictogramElement** contém a representação gráfica do objeto.

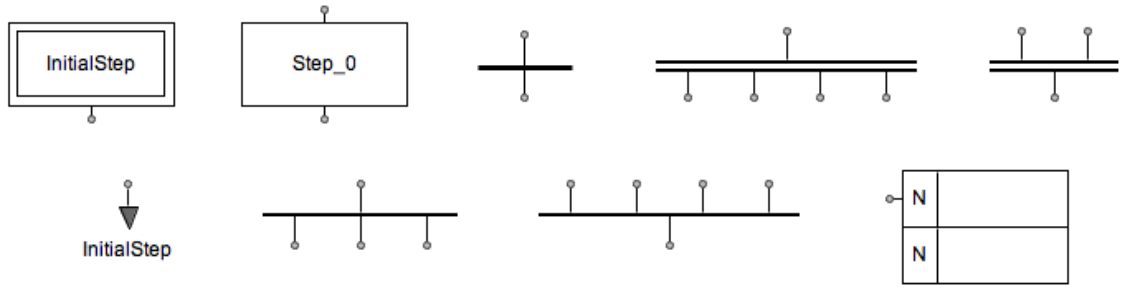


Figura 3.12: Objetos gráficos SFC criados.
Da esquerda para a direita, e de cima para baixo: *Initial Step*; *Step*; *Transition*; *Simultaneous Divergence*; *Simultaneous Convergence*; *Jump Step*; *Selection Divergence*; *Selection Convergence* e *Action Block*.

3.2.2.3 *Create Connection Feature* e *Add Connection Feature*

A classe responsável pela *Create Connection Feature* contém um método *canStartConnection()*, que verifica se a conexão que se pretende iniciar começa num dos objetos SFC. Retorna *true* em caso afirmativo, e *false* em caso negativo. Contém também um método *canCreate()*, que verifica se o a conexão que se pretende criar é válida (ver abaixo), e retorna *true* em caso afirmativo, e *false* em caso negativo.

No SFC nem todos os blocos se podem ligar o todos os outros. Um *Step* deve ter conectado a montante uma *Transition*, uma *Simultaneous Divergence* ou uma *Selection Convergence*, e a jusante uma *Transition*, uma *Selection Divergence*, uma *Simultaneous Convergence* ou um *Jump Step*. Já as *Transitions* deve ter conectado a montante um *Step*, uma *Simultaneous Convergence* ou uma *Selection Divergence*, e a jusante um *Step*, uma *Selection Convergence*, uma *Simultaneous Divergence* ou um *Jump Step*. No caso das ramificações, as *Simultaneous Convergence* e as *Selection Divergence* devem ter conectados *Steps* a montante e *Transitions* ou *Jump Steps* a jusante. As *Selection Convergence* e as *Simultaneous Divergence* devem ter conectadas *Transitions* a montante e *Steps* ou *Jump Steps* a jusante. As *Actions Blocks* apenas se podem conectar lateralmente a *Steps*. Para proceder à verificação da validade das conexões foram criados os seguintes métodos:

- isStep2ActionBlock(sourceObject, targetObject);
- isActionBlock2Step(sourceObject, targetObject);
- isBottomOfTransition2TopOfStep(sourceObject, targetObject);
- isTopOfTransition2BottomOfStep(sourceObject, targetObject);
- isBottomOfTransition2TopOfSimultaneousDivergence(sourceObject, targetObject);
- isTopOfTransition2BottomOfSimultaneousConvergence(sourceObject, targetObject);
- isTopOfTransition2BottomOfSelectionDivergence(sourceObject, targetObject);

- isBottomOfTransition2TopOfSelectionConvergence(sourceObject, targetObject);
- isTopOfStep2BottomOfTransition(sourceObject, targetObject);
- isBottomOfStep2TopOfTransition(sourceObject, targetObject);
- isTopOfStep2BottomOfSimultaneousDivergence(sourceObject, targetObject);
- isBottomOfStep2TopOfSimultaneousConvergence(sourceObject, targetObject);
- isBottomOfStep2TopOfSelectionDivergence(sourceObject, targetObject);
- isTopOfStep2BottomOfSelectionConvergence(sourceObject, targetObject);
- isTopOfSimultaneousDivergence2BottomOfTransition(sourceObject, targetObject);
- isBottomOfSimultaneousDivergence2TopOfStep(sourceObject, targetObject);
- isBottomOfSimultaneousConvergence2TopOfTransition(sourceObject, targetObject);
- isTopOfSimultaneousConvergence2BottomOfStep(sourceObject, targetObject);
- isTopOfSelectionDivergence2BottomOfTransition(sourceObject, targetObject);
- isBottomOfSelectionDivergence2TopOfStep(sourceObject, targetObject);
- isTopOfSelectionConvergence2BottomOfTransition(sourceObject, targetObject);
- isBottomOfSelectionConvergence2TopOfStep(sourceObject, targetObject);

A classe contém também um método *create()*, que cria um objeto do tipo *Connection*, usando a classe **SequentialFunctionChartFactory**, e guarda-o no modelo. De seguida chama a *Add Feature* responsável pelo desenho das conexões (**AddConnection**) para que esta seja inserida no diagrama.

A classe responsável pela *Add Connection Feature* contém um método *canAdd*, que verifica se o objeto que se quer inserir é do tipo *Connection*, e retorna *true* em caso afirmativo, e *false* em caso negativo. O método *add()* da mesma classe usa os interfaces *IGaService* e *IPeService* para adicionar uma linha à conexão. Se necessário adiciona pontos de quebra à conexão por forma a que esta contenha apenas linhas horizontais e verticais. Por fim atualiza o modelo de ligação com o método *link(PictogramElement pe, Object businessObject)*.

3.2.2.4 *Create Features*

Cada uma das classes responsáveis pelas *Create Features* contém um método *canCreate()*, que apenas verifica se o local onde se quer inserir o objeto é um diagrama, e retorna *true* em caso afirmativo, e *false* em caso negativo. Contém ainda o método *create()*, que cria o respetivo objeto, usando a classe do metamodelo Java **SequentialFunctionChartFactory**, e atualiza o modelo de ligação através do método *addGraphicalRepresentation()*.

3.2.2.5 *Update Features*

Alguns objetos gráficos podem sofrer alterações no seu desenho ao longo da criação de um diagrama (*e.g.*, pode ser necessário adicionar um ponto de conexão a um elemento *Selection Divergence*). Quando o utilizador requer esta alteração (ver *Custom Features* abaixo), o respetivo elemento do modelo é alterado. Quando o Graphiti deteta uma alteração a um qualquer elemento do modelo, ele corre todas as classes que façam a atualização dos elementos gráficos: as classes responsáveis pelas *Update Features*. No nosso diagrama tivemos necessidade de criar *Update Features* para os elementos *Selection Divergence* (**UpdateSelectionDivergence**), *Selection Convergence* (**UpdateSelectionConvergence**), *Simultaneous Divergence* (**UpdateSimultaneousDivergence**), *Simultaneous Convergence* (**UpdateSimultaneousConvergence**) e ainda para o elemento *Action Block* (**UpdateActionBlock**).

Cada *Update Feature* contém um método *canUpdate()*, que verifica se o objeto que se quer atualizar é do tipo correto (*e.g.*, do tipo *Action Block* na classe **UpdateActionBlock**), e retorna *true* em caso afirmativo, e *false* em caso negativo. Contém também o método *updateNeeded()*, que verifica se o objeto necessita ser atualizado. Nas nossas classes isso acontece sempre que, pela ação do utilizador, o objeto do modelo sofre alterações e essas alterações precisem de ser refletidas nos objetos gráficos: *e.g.*, se o utilizador inserir mais um ponto de conexão num objeto *Selection Divergence*, através do menu de contexto, essa alteração é feita ao objeto do modelo. Quando a *Update Feature* do objeto *Selection Divergence* correr, ela vai verificar se o objeto do modelo e o objeto gráfico têm o mesmo numero de pontos de conexão: se não tiverem, é necessário atualizar o objeto gráfico.

Cada classe contém ainda o método *update()*, que trata da atualização do objeto gráfico. Em cada um destes métodos o objeto gráfico é desligado do objeto do modelo (é removido do modelo de ligação), e é apagado. É depois criado outro objeto gráfico no seu lugar (através da respetiva *Add Feature*), e ligado ao objeto do modelo em questão. Por fim, as conexões ao objeto antigo são também apagadas (através da respetiva *Delete Feature*), e são criadas conexões novas no seu lugar (através da *Add Connection Feature*).

3.2.2.6 *Move Shape Feature*

De cada vez que um objeto gráfico é movido, caso este esteja conectado a outro objeto, essa conexão tem de ser recriada. O Graphiti redesenha automaticamente a conexão, mas como uma linha reta entre os dois objetos. No nosso diagrama nós queremos que todas conexões sejam compostas apenas por linhas horizontais ou verticais, mas o Graphiti não permite assegurar esse comportamento. Por esse motivo temos de voltar a redesenhar as conexões nós mesmos, inserindo, se necessário, alguns pontos de quebra no interior das mesmas.

O nosso projeto contém apenas uma *Move Shape Feature*, que é chamada quando qualquer objeto gráfico é movido, e serve apenas para redesenhar as conexões que lhe estão associadas. Contém um método *moveShape()* que verifica se as conexões ao objeto, depois deste ser movido, se mantêm linhas horizontais ou verticais: se não for o caso as conexões têm de ser alteradas. As

alterações processam-se segundo a seguinte lógica: caso seja uma conexão direta (sem pontos de quebra) são-lhe inseridos dois pontos de quebra, ficando a conexão composta por três linhas retas horizontais e verticais. Caso já existam pontos de quebra, eles são movidos por forma a que as três linhas da conexão se mantenham horizontais e verticais (ver Figura 3.13).

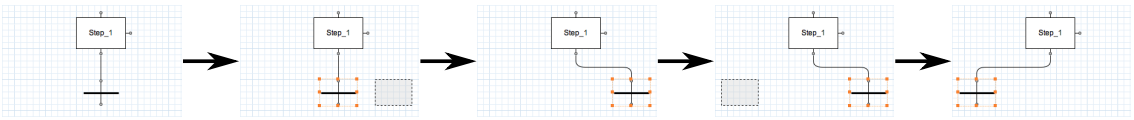


Figura 3.13: Alteração das conexões ao mover objeto gráfico.

3.2.2.7 *Move Bendpoint Feature* e *Add Bendpoint Feature*

Por forma a dar alguma liberdade ao utilizador na manipulação das conexões, permitimos que o utilizador arraste uma conexão verticalmente. Para isso basta que ele mova um dos seus pontos de quebra. Pode ver na Figura 3.14 o funcionamento desse processo. Para isso criámos uma *Move Bendpoint Feature*, que redesenha a conexão com base na nova posição do nó deslocado. Essa classe contém o método *execute()*, que usa a informação da posição para a qual o ponto de quebra foi movido, e move o outro ponto de quebra a mesma distância, por forma a manter todas as linhas da conexão horizontais ou verticais. Caso o ponto de quebra seja movido diagonalmente, o deslocamento horizontal é anulado.

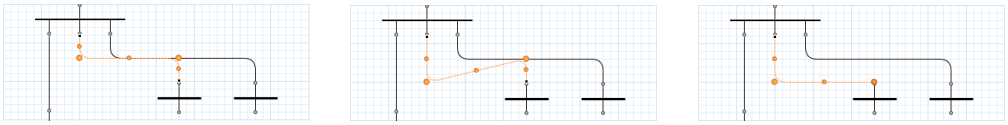


Figura 3.14: Mover uma conexão verticalmente.

No nosso editor gráfico, quando uma classe precisa de inserir um ponto de quebra numa conexão, ela insere-o diretamente: o objecto que representa uma conexão contém uma lista com todos os pontos de quebra, e a conexão é desenhada conectando esses pontos. No entanto, uma conexão Graphiti contém, para além dos pontos de quebra, uns *handlers* que permitem criar novos pontos de quebra, distorcendo a conexão. O Graphiti insere um *handler* entre cada dois pontos de quebra da conexão, e representa-os por uns círculos pequenos — os pontos de quebra são um pouco maiores. Pode ver-se na Figura 3.15 o que acontece à conexão quando um desses *handlers* é movido. Nos nossos diagramas não queremos que o utilizador possa inserir um ponto de quebra dessa forma. Por isso criamos uma *Add Bendpoint Feature*, e nela fazemos *override* aos métodos *addBendpoint()* e *excute()*. No corpo desses métodos não inserimos qualquer código. Assim, quando o utilizador move um *handler* numa conexão, a conexão não se altera.

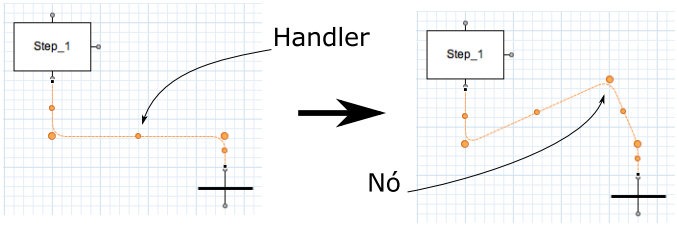


Figura 3.15: Inserção indesejada de ponto de quebra ao mover um *handler*.

3.2.2.8 *Resize Shape Feature*

Os diagramas padrão do Graphiti permitem que o utilizador aumente ou diminua o tamanho de um elemento gráfico. Para isso insere *handlers* no contorno da caixa de seleção do objeto. Caso o utilizador não escreva as *features* necessárias para efetuar o redimensionamento dos objetos, eles não são alterados, mas a caixa de seleção é-o (ver Figura 3.16). Como não pretendemos que o utilizador consiga alterar os objetos gráficos, não precisámos escrever as respetivas *features*. No entanto, o comportamento descrito atrás também não é desejado: por isso criámos uma *Resize Shape Feature*, e nela desabilitamos as funções de redimensionamento padrão do Graphiti para todos os objetos gráficos.

Figura 3.16: Comportamento indesejado no redimensionamento do Graphiti.

Essa classe contém o método *canResizeShape()*, que retorna sempre *false*. Desse modo o Graphiti não permite que nenhum elemento gráfico seja redimensionado.

3.2.2.9 *Delete Feature*

Quando o utilizador quer apagar um elemento gráfico do editor, as *Remove Features* do Graphiti conseguem tratar do processo. O Graphiti assegura também que as respetivas *Delete Features* são executadas para eliminar os respetivos elementos do modelo.

No entanto, o comportamento das *Delete Features* para com o utilizador não é o desejado: mostra uma mensagem de confirmação sempre que se quer apagar um elemento, o que se mostrou um pouco incomodativo. Para corrigir este problema criámos uma *Delete Feature*, e nela chamámos o construtor da sua superclasse, para que o processo não seja interrompido. No entanto fazemos *override* ao método *getUserDecision()*: por omissão, este método apresenta uma janela de diálogo ao utilizador a confirmar a eliminação do elemento. Como não queremos que essa janela apareça, simplesmente retornámos *true* neste método.

3.2.2.10 *Custom Features*

Para além das *features* anteriores, tivemos de criar algumas para as quais não existem *Default Features*. Essas *features* — *Custom Features* —, são usadas para alterar o número de pontos de conexão nos objetos gráficos *Simultaneous Divergence*, *Simultaneous Convergence*, *Selection Divergence* e *Selection Convergence*. São também usadas para aumentar o número de ações num *Action Block*. Todas elas requerem que o elemento gráfico em questão seja redesenhado, e por isso usam as respetivas *Add Features*. As classes que implementam estas *Custom Features* são: **IncreaseConnections** (aumenta o número de pontos de conexão de uma ramificação), **DecreaseConnections** (diminui o número de pontos de conexão de uma ramificação), **SetConnections** (altera para um valor inserido pelo utilizador o número de pontos de conexão de uma ramificação), e **IncreaseActions** (aumenta o número de ações de um *Action Block*).

Cada uma destas classes contém um método *canExecute()*, que verifica se o objeto que se quer alterar é do tipo correto (*i.e.*, do tipo *Action Block* na classe **IncreaseActions**, ou dos tipos *Selection Divergence*, *Selection Convergence*, *Simultaneous Divergence* ou *Simultaneous Convergence* nas classes **IncreaseConnections**, **DecreaseConnections** e **SetConnections**). Retorna *true* em caso afirmativo, e *false* em caso negativo.

O método *execute()* da classe **IncreaseActions** aumenta o número de ações no objeto do modelo (*ActionBlock*), e chama a *Update Feature* respetiva (**UpdateActionBlock**) para redesenhar o objeto gráfico.

O método *execute()* da classe **IncreaseConnections** aumenta o número de pontos de conexão no objeto do modelo, e chama a *Update Feature* respetiva.

O método *execute()* da classe **DecreaseConnections** verifica se o número de pontos de conexão é superior a dois, e se for o caso diminui o número de pontos de conexão do objeto do modelo. De seguida chama a respetiva *Update Feature*. De notar que não é possível apagar pontos de conexão que tenham uma conexão. Se não houver pontos de conexão livres, não é possível diminuir o número de pontos de conexão do objeto. Nesse caso deve-se primeiro apagar pelo menos uma conexão.

O método *execute()* da classe **SetConnections** mostra uma caixa de diálogo onde o utilizador pode inserir o número de pontos de conexão que quer para o objeto. Os pontos de conexão são criados se o utilizador inserir um número válido: se for maior que dois e maior ou igual ao número de pontos de conexão em uso.

3.2.3 Janelas de Diálogo

Como vimos na Secção 3.2.2, o nosso editor gráfico usa janelas de diálogo para interagir com o utilizador. Para isso foram criadas as classes **PromptStepName** e **PromptNumberOfConnections**. Ambas estendem a classe **TitleAreaDialog**, que cria uma janela com um espaço reservado para fornecer feedback ao utilizador. O texto a inserir nesse espaço é definido com os métodos *setMessage()* ou *setErrorMessage()*. Se usarmos o método *setMessage()* a mensagem é acompanhada

de um sinal de informação, se usarmos o método *setErrorMessage()* a mensagem é acompanhada de um sinal de erro.

A classe **PromptStepName** é usada para pedir ao utilizador o nome para um *Step*/*Initial Step* a inserir no diagrama. É instanciada nas classes **AddStep** e **AddInitialStep**.

A classe **PromptNumberOfConnections** é usada para pedir ao utilizador o número de pontos de conexão a criar num objeto. É instanciada na classe **SetConnections**.

Pode ver duas instâncias destas janelas de diálogo nas Figuras 3.17 e 3.18.

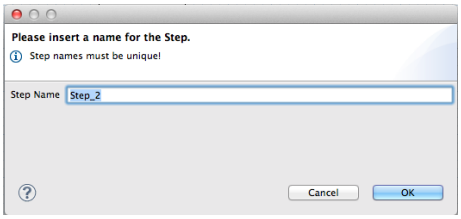


Figura 3.17: Janela de diálogo **PromptStepName**.

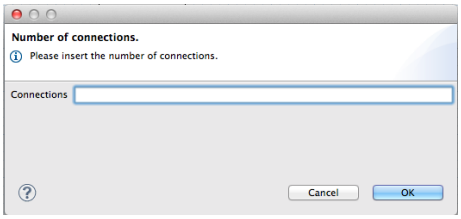


Figura 3.18: Janela de diálogo **PromptNumberOfConnections**.

3.2.4 Conversão para XML

A conversão para XML foi programada por forma a gerar ficheiros que respeitem o esquema XML definido em *http://www.plcopen.org/xml/tc6_0201*. São traduzidos para XML todos os elementos gráficos.

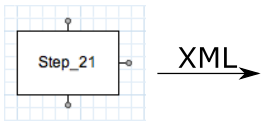
A classe que efetua a conversão para XML chama-se **SFC2XML**. Nela são executados em sequência os métodos *loadObjects()*, *loadConnections()*, *writeXML()*, *createDummyProject()* e *saveToFile()*.

No método *loadObjects()* são lidos todos os elementos do modelo, e é-lhes atribuído individualmente um identificador único, no campo *localId* — campo existente em todos os objetos SFC (ver Secção 3.2.1). É-lhes também atribuída a posição do objeto gráfico a que estão associados.

No método *loadConnections()* são lidas todas as conexões do diagrama, e as coordenadas de todos os seus nós são guardadas nos objetos a que estão conectadas. É também aqui que são criadas as referências cruzadas nos objetos ligados por uma conexão.

Os métodos anteriores servem apenas para preparar os objetos por forma a poderem ser processados pelo método *writeXML()*. Nele são de novo lidos todos os objetos do modelo, já com as alterações feitas nos métodos anteriores, e é escrito um bloco de texto com o XML necessário para cada objeto. Esses blocos são depois agrupados numa única *String* — *xml*. Pode ver na Figura 3.19 um *Step* e o respetivo bloco de texto para ser inserido no ficheiro XML.

As informações sobre as dimensões dos objetos e as posições relativas dos pontos de conexão são guardadas numa classe à parte, para mais fácil edição: a classe **SFCObjectData**. Quando as *Add Features* criam os objetos elas vêm a esta classe ler a informação sobre as suas dimensões e as posições onde devem colocar as âncoras.




```
<step localId="13" height="51" width="81" name="Step_21" initialStep="false" negated="false" >
  <position x="70" y="200"/>
  <connectionPointIn>
    <relPosition x="40" y="0"/>
  </connectionPointIn>
  <connectionPointOut formalParameter="">
    <relPosition x="40" y="51"/>
  </connectionPointOut>
  <connectionPointOutAction formalParameter="">
    <relPosition x="81" y="25"/>
  </connectionPointOutAction>
</step>
```

Figura 3.19: *Step* e respetivo bloco XML.

O nosso editor gráfico cria o corpo de um POU. Para que seja possível importar o ficheiro XML num outro IDE, ele deve representar um projeto completo, e não apenas o corpo de um POU. Desse modo, no método *createDummyProject()* é inserida à *String xml* um *header* e um *footer* com informação de um projeto fictício. Pode ver na Figura 3.20 um ficheiro XML gerado de um diagrama contendo apenas um *Initial Step*. Na imagem estão identificados o *header*, o *footer* e o texto criado pelo nosso conversor.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <project xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://www.plcopen.org/xml/tc6.xsd"
   xmlns:xhtml="http://www.w3.org/1999/xhtml" xsi:schemaLocation="http://www.plcopen.org/xml/tc6.xsd">
3    <fileHeader companyName="FEUP" productName="SFC-Test-Product" productVersion="1" creationDateTime="2014-04-01T18:19:20"/>
4    <contentHeader name="SFC-test" modificationDateTime="2014-04-01T19:20:21">
5      <coordinateInfo>
6        <fbd>
7          <scaling x="0" y="0"/>
8        </fbd>
9        <ld>
10         <scaling x="0" y="0"/>
11       </ld>
12       <sfc>
13         <scaling x="0" y="0"/>
14       </sfc>
15     </coordinateInfo>
16   </contentHeader>
17   <types>
18     <dataTypes />
19     <pous>
20       <pou name="SFC_Teste_POU" pouType="functionBlock">
21         <interface />
22         <body>
23           <SFC>
24             <step localId="0" height="51" width="81" name="InitialStep" initialStep="true" negated="false" >
25               <position x="250" y="60"/>
26               <connectionPointIn>
27                 <relPosition x="40" y="0"/>
28                 <connection refLocalId="3">
29                 </connection>
30               </connectionPointIn>
31               <connectionPointOut formalParameter="">
32                 <relPosition x="40" y="51"/>
33               </connectionPointOut>
34               <connectionPointOutAction formalParameter="">
35                 <relPosition x="81" y="25"/>
36               </connectionPointOutAction>
37             </step>
38           </SFC>
39         </body>
40       </pou>
41     </pous>
42   </types>
43   <instances>
44     <configurations />
45   </instances>
46 </project>
```

HEADER

XML GERADO

FOOTER

Figura 3.20: Ficheiro SFC convertido em XML.

Finalmente no método *saveToFile()* é apresentada ao utilizador uma janela de diálogo onde ele pode escolher o nome e o local onde quer guardar o ficheiro XML. De seguida esse ficheiro é criado e é-lhe escrita a *String xml*. Por fim o ficheiro é fechado.

3.2.5 Conversão para texto

A conversão do diagrama SFC para texto processa-se de forma semelhante à conversão para XML, embora a informação relativa às posições dos elementos, incluindo dos nós das conexões, não seja utilizada. Outra diferença reside no facto de nos ficheiros textuais de SFC apenas aparecerem objetos do tipo *Step*, *Initial Step* e *Transition*. Os *Jump Steps* são vistos como uma conexão que liga o objeto a montante ao objeto ao qual faz referência. Os objetos do tipo *Action Block* também não aparecem nos ficheiros de texto, mas as ações que os compõem aparecem como blocos do tipo *Action*. Os restantes objetos são também vistos como conexões que ligam os objetos a montante com os objetos a jusante.

Para realizar a conversão foi criada a classe **SFC2Text**, onde são executados, por ordem, os métodos *loadObjects()*, *loadConnections()*, *writeText()*, *createDummyProject()* e *saveToFile()*.

O método *loadObjects()* faz o mesmo processamento que o mesmo método com o mesmo nome da classe **SFC2XML** (ver Secção 3.2.4).

O método *loadConnections()* lê todas as conexões do diagrama, e cria as referências cruzadas nos objetos ligados por cada conexão. No entanto, se uma conexão ligar um objeto que não seja um *Step*, *Initial Step*, *Transition*, ou *Action Block*, as referências cruzadas são criadas de modo a ignorarem esse mesmo objeto. Pode ver na Figura 3.21 um exemplo de como um diagrama é visto pelo método *loadConnections()*.

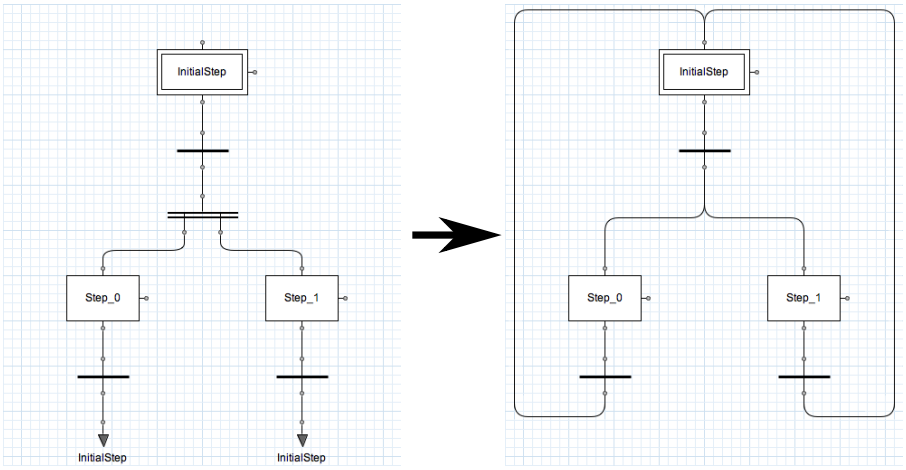


Figura 3.21: Como um diagrama é visto pelo método *loadConnections()* da classe **SFC2Text**.

O método *writeText()* lê os objetos do diagrama à procura de um *Initial Step*. Este deve ser o primeiro elemento do ficheiro de texto. De seguida lê os restantes objetos, ignorando aqueles que não são representados na representação textual do diagrama, e cria blocos de texto para cada um deles. Esses blocos são depois agrupados numa *String — text*.

O método *createDummyProject()*, tal como explicado na Secção 3.2.4, cria um *header* e um *footer* para replicar um projeto completo. Pode ver na Figura 3.22 um ficheiro de texto gerado de um diagrama contendo apenas um *Initial Step*. Na imagem estão identificados o *header*, o *footer* e o texto criado pelo nosso conversor.

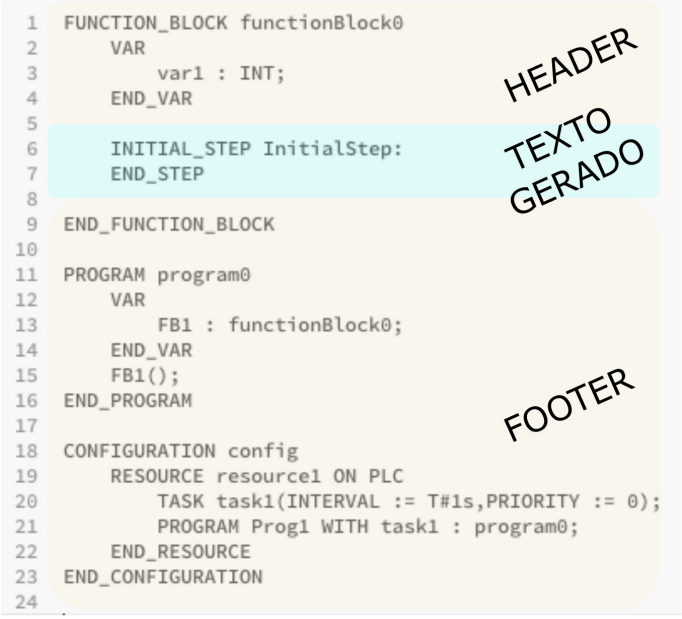


Figura 3.22: Ficheiro SFC convertido em texto.

Finalmente no método *saveToFile()* é apresentada ao utilizador uma janela de diálogo onde ele pode escolher o nome e o local onde quer guardar o ficheiro de texto. De seguida esse ficheiro é criado e é-lhe escrita a *String text*. Por fim o ficheiro é fechado.

Capítulo 4

Testes e Conclusões

Dos objetivos que inicialmente nos propusemos realizar, começámos por implementar os editores textuais, e de seguida implementamos o editor gráfico para SFC. Os restantes blocos auxiliares não chegaram a ser implementados, pois o tempo não foi suficiente.

O desenvolvimento dos editores textuais para *Structured Text* e *Instruction List* foi executada em paralelo, devido a algumas semelhanças entre eles. Depois de criados, os editores foram submetidos aos testes que a seguir se descrevem.

4.0.6 Teste à *Syntax Highlight*

Foi criado um ficheiro no editor de ST, e nele foram escritas todas as *keywords* da linguagem *Structured Text*, bem como todos os tipos de dados utilizados na linguagem. Foram também inseridos blocos com comentários. A *syntax highlight* foi feita corretamente em todos os casos, como se pode ver na Figura 4.1.

Do mesmo modo, foi criado um ficheiro no editor de IL, e nele foram escritas todas as *keywords* da linguagem *Instruction List*, bem como todos os tipos de dados utilizados na linguagem e blocos com comentários. A *syntax highlight* foi feita corretamente em todos os casos, como se pode ver na Figura 4.1.

4.0.7 Teste à gravação e carregamento de ficheiros textuais

Para testar a gravação em disco dos ficheiros ST e IL, foram criados vários ficheiros, e foram gravados através do menu Eclipse. De seguida foram fechados e procurou-se no sistema de ficheiros do computador pelos respetivos ficheiros, que foram encontrados.

Para testar o carregamento dos ficheiros pelo Eclipse, efetuámos duplo clique, no *Project Explorer*, em vários ficheiros ST e IL previamente criados. O Eclipse abriu cada um deles nos respetivos editores. Pode ver na Figura 4.2 o resultado deste teste. Na mesma figura pode-se verificar a correta atribuição dos ícones aos ficheiros ST e IL.

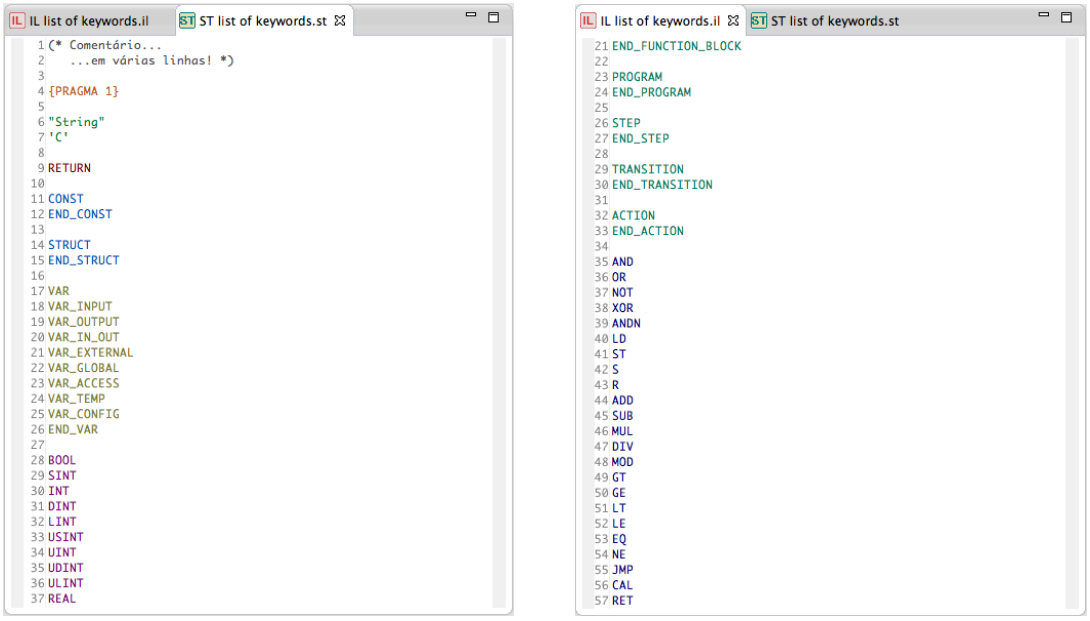


Figura 4.1: *Syntax Highlight* em ficheiros ST e IL.

4.0.8 Teste à abertura de ficheiros ST e IL noutros editores

Os ficheiros resultantes da gravação dos projetos escritos nos editores textuais podem ser abertos e editados externamente. Para testar esta funcionalidade abrimos vários ficheiros ST e IL em editores de texto padrão. Foram todos abertos e editados com sucesso. Pode ver-se na Figura 4.3 o resultado de um desses testes.

4.0.9 Teste à gravação dos documentos em ST e IL para XML

A conversão dos documentos em IL e ST para XML é feita copiando o texto do editor para um projeto fictício. Dessa forma a importação de projetos ST e IL por outros IDE's, depois da conversão para XML é apenas dependente da estrutura do código escrito pelo utilizador. De qualquer forma foi testada a conversão criando ficheiros de teste em ST e IL, e estes foram abertos com sucesso no IDE Beremiz. Pode ver na Figura 4.4 o editor com o texto em ST usado num desses testes, e na Figura 4.5 o mesmo ficheiro aberto no IDE Beremiz. A realização dos testes com o Beremiz foi feita de acordo com o explicado na Secção 4.0.11.

Depois de criados os editores textuais, iniciou-se o desenvolvimento dos editores gráficos. Aqui foram encontradas algumas dificuldades, pois a parte da API Eclipse para criação de editores gráficos não está tão desenvolvida como a parte da API para criação de editores textuais. O GEF permite bastante versatilidade, mas é de bastante baixo nível, e não teria sido possível chegar ao

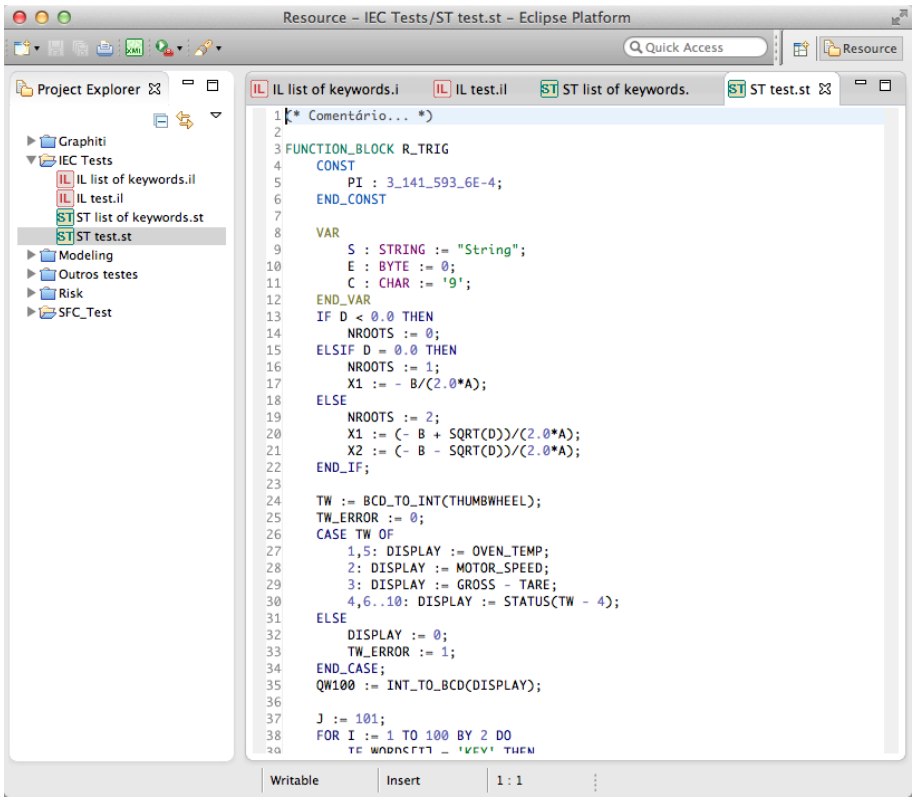


Figura 4.2: Teste à abertura de ficheiros ST e IL.

nível de desenvolvimento que conseguimos, no tempo que tínhamos disponível, caso continuassemos o desenvolvimento dos editores gráficos com GEF. Foi por isso fundamental a escolha da *framework* Graphiti para o desenvolvimento dos editores. O uso do Graphiti trouxe, no entanto, alguns problemas, que só podem ser resolvidos alterando o seu código fonte (*e.g.*, como já foi dito, o Graphiti não permite a implementação direta do roteamento pretendido). Por outro lado, A utilização do Graphiti permite que no editor SFC desenvolvido seja fácil o alinhamento dos blocos, permitindo a visualização de uma grelha, bem como a visualização de guias.

Desenvolvemos, então, um editor gráfico para a linguagem SFC. O editor não está completo, embora já contenha todos os blocos da linguagem, e permita a conversão para XML e para texto. Foram-lhe realizados os testes que a seguir se descrevem.

4.0.10 Teste à criação de diagramas complexos

Por forma a testar o alinhamento e a possibilidade de criar diagramas complexos, contendo todos os tipos de elementos gráficos foram criados alguns diagramas de teste. Pode ver na Figura 4.6 um desses diagramas. O editor contém, no entanto, alguns problemas no que respeita à manipulação de conexões. Se o utilizador mover os elementos por forma a que as âncoras de saída fiquem acima das âncoras de entrada, a conexão não fica visualmente agradável. Também se o utilizador mover os pontos de quebra para cima de uma âncora o Graphiti elimina o ponto de

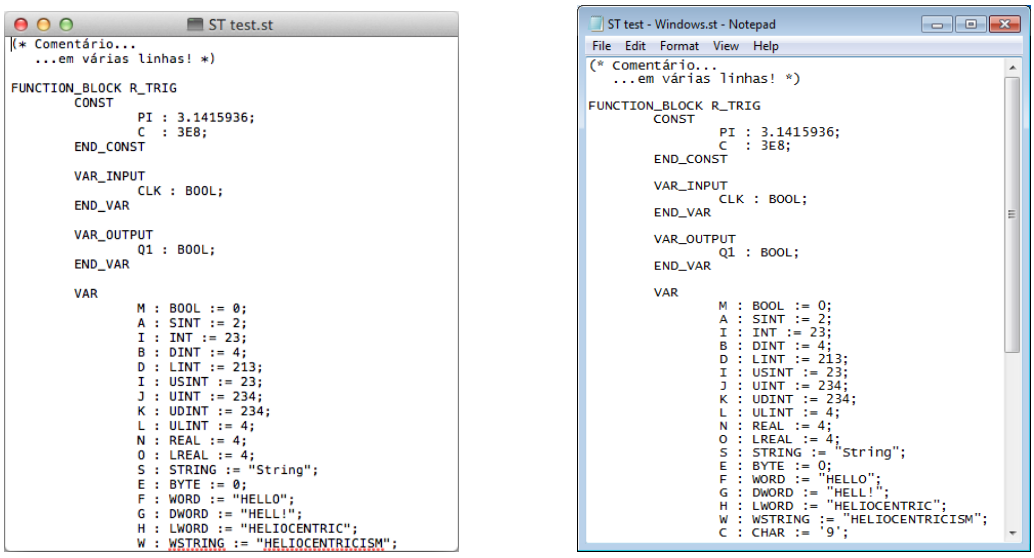


Figura 4.3: Teste à abertura externa de ficheiros ST e IL.

quebra, deixando a conexão de consistir apenas de linhas verticais e horizontais. Pode ver estes problemas na Figura 4.7.

4.0.11 Testes à conversão para XML

Por forma a testar a conversão dos diagramas para XML foram criados diagramas de complexidades variadas. Foram depois importados nos IDE's Beremiz e CoDeSys.

O IDE Beremiz guarda todos os projetos em formato XML, por isso, para testarmos o nosso projeto SFC basta substituir o ficheiro adequado, numa pasta onde já se encontre um projeto Beremiz. Para isso abrimos o Beremiz e criámos um novo projeto. De seguida fechámos o projeto, fomos até à pasta onde o projeto está guardado, e substituímos o ficheiro *plc.xml* pelo nosso ficheiro XML (que deve ter o mesmo nome). Depois de voltarmos ao Beremiz abrimos de novo o projeto criado anteriormente, e o nosso POU já se encontra presente e pode ser visualizado. Pode ver na Figura 4.8 um ficheiro criado no nosso editor aberto no editor Beremiz. O ficheiro utilizado pode ser visto no nosso editor na Figura 4.9. Quando aberto no Beremiz o diagrama apresenta algumas distorções nas conexões, que acreditamos se ficarem a dever à diferença nas dimensões dos objetos utilizadas por nós e pelo Beremiz.

Para testarmos os nossos diagramas convertidos em XML no IDE CoDeSys tivemos de criar um novo projeto do tipo *Standard Project*. De seguida selecionamos no menu *Devices* o item *Application* e selecionamos a opção *Import PLCOpenXML* no menu *Project*. O nosso POU é de seguida importado e inserido na aplicação criada por omissão pelo CoDeSys.

Pode ver na Figura 4.10 um ficheiro criado no nosso editor importado para pelo CoDeSys. O ficheiro utilizado pode ser visto no nosso editor na Figura 4.9. Ao contrário do IDE Beremiz,

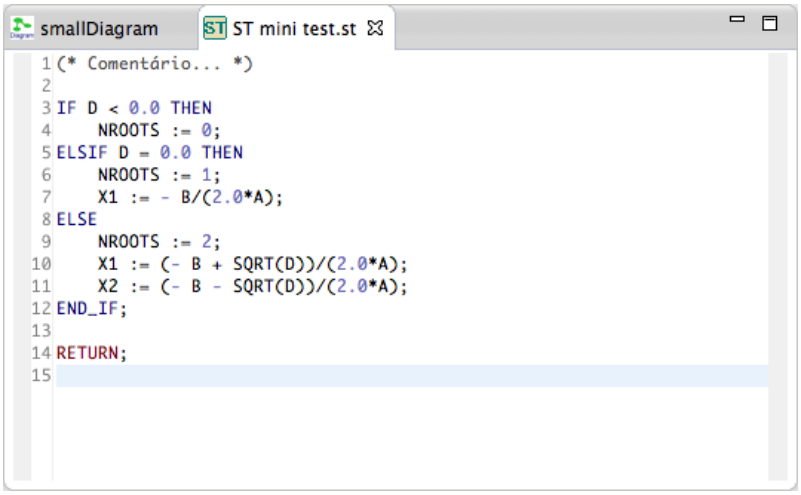


Figura 4.4: Código ST para conversão em XML.

o CoDeSys não utiliza a informação dos tamanhos e posições dos nossos objetos gráficos, embora essa informação esteja presente no ficheiro XML. Ele estrutura automaticamente o diagrama apenas com base nos objetos e nas ligações entre eles.

4.0.12 Teste à conversão para texto

O nosso editor permite ainda converter um diagrama SFC para texto, por forma a este ser compilado com o MATIEC. Para testarmos esta funcionalidade criámos alguns diagramas, e efetuámos a conversão para texto. De seguida utilizamos o compilador MATIEC para compilar os ficheiros resultantes. Os testes feitos foram todos bem sucedidos. Pode ver o resultado de um desses testes na Figura 4.11. O diagrama usado nesse teste pode ser visto na Figura 4.12, e o ficheiro de Texto resultante da conversão pode ser visto na Figura 4.13.

4.1 Trabalho Futuro

O nosso IDE pode ainda ser desenvolvido, sendo possível adicionar-lhe algumas funcionalidade importantes. Segue-se a discussão de algumas dessas funcionalidades.

4.1.1 Integração do compilador MATIEC

Futuramente pode ser feita a integração do compilador *MATIEC* no IDE. Essa integração permitiria a compilação dos programas diretamente do IDE, não tendo o utilizador de recorrer a ferramentas externas. O IDE Beremiz faz já essa integração com bons resultados.

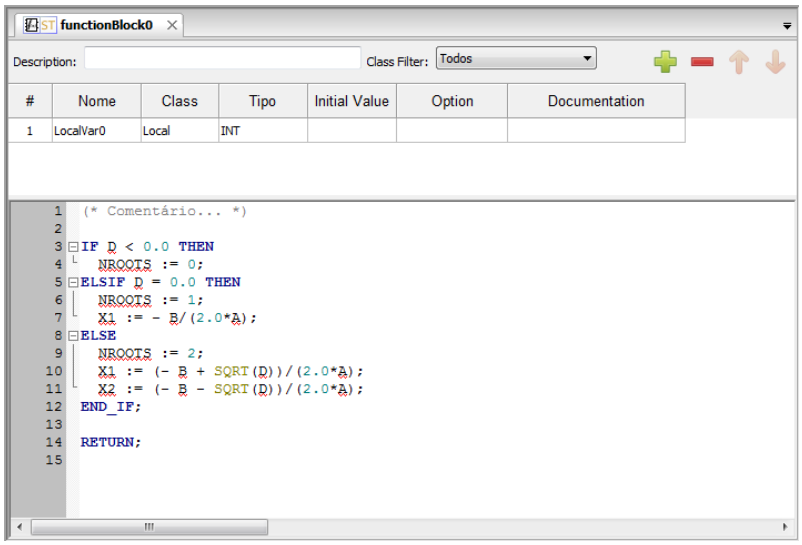


Figura 4.5: Ficheiro ST convertido em XML e aberto no Beremiz.

4.1.2 Melhoramentos nos editores textuais

Os editores textuais podem futuramente ser melhorados em alguns aspectos, como por exemplo, com a implementação de *code folding* e *code completion*. A falta destas funcionalidades começa a sentir-se, principalmente em programas longos.

4.1.3 Melhoramentos nos editores gráficos

Num desenvolvimento futuro será possível a criação de metamodelos e editores para as restantes linguagens gráficas (*Function Block Diagram* e *Ladder Diagram*). Esses metamodelos e editores conterão uma estrutura semelhante ao metamodelo e editor para SFC.

O roteamento que o Graphiti disponibiliza não é o ideal para o desenvolvimento de diagramas nas linguagens da norma IEC 61131, pois não permite criar cantos com ângulos retos (o Graphiti arredonda automaticamente os cantos). A única maneira de resolver o problema passaria pela alteração do código fonte do Graphiti, juntando à classe que cria os cantos um parâmetro que indicasse o raio de curvatura desejado. Depois de feitas as alterações à classe, ela poderia ser submetida à Eclipse Foundation com o intuito de ser integrada na *framework* Graphiti. Quando contactada por nós, a Eclipse Foundation mostrou-se recetiva à integração dessa funcionalidade na *framework*, pelo que esse desenvolvimento pode ser feito futuramente.

As conexões por nós criadas podem ainda ser melhoradas, nomeadamente no caso em que o objeto a jusante da conexão fica acima do objeto a montante, e no caso do desaparecimento de um ponto de quebra. Atualmente as conexões que resultam neste tipo de situação não são visualmente agradáveis.

Futuramente pode também ser implementada a funcionalidade de redimensionar alguns dos objetos gráficos SFC. Essa funcionalidade seria útil, principalmente, nos objetos *Step* e *Action Block*, que contêm no seu interior, texto que pode ser mais ou menos longo. Outra funcionalidade

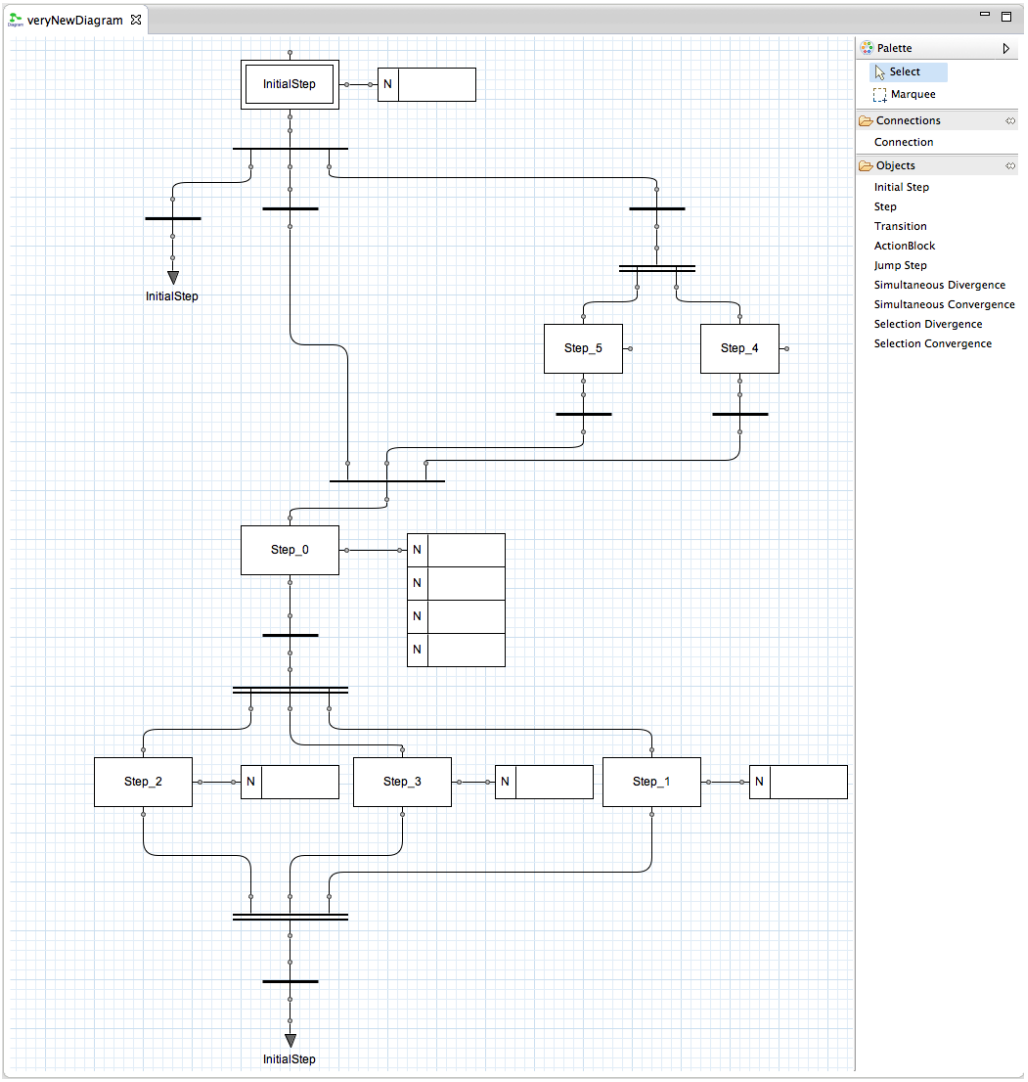


Figura 4.6: Diagrama SFC de teste.

relacionada, seria a possibilidade de o utilizador poder escolher quais os tamanhos por omissão de todos os elementos gráficos.

Um aspeto importante a ser melhorado nos editores gráficos é a possibilidade de se criarem *Actions* e *Conditions*. Sem estes blocos o editor fica bastante limitado.

A barra lateral onde estão listados os objetos a inserir no diagrama pode também ser melhorada com a inclusão de ícones para cada um dos objetos.

4.1.4 Criação de blocos auxiliares

Para trabalho futuro ficaram ainda os blocos auxiliares do nosso IDE: *view* com os tipos de variáveis, *view* com a lista de POU's criados pelo utilizador, vários *wizards* para criação de projetos, entre outras possíveis.

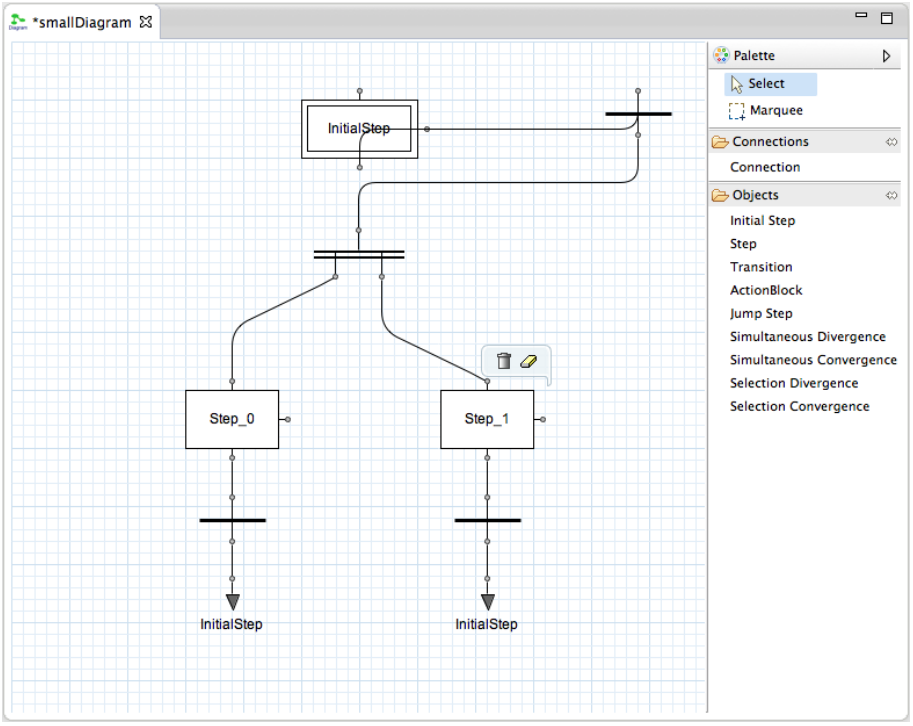


Figura 4.7: Problemas nas conexões dos diagramas SFC.

4.1.5 Criação de Projetos

Por fim, pode também ser criado futuramente um *plug-in* para criar um projeto, e dentro desse projeto poderão depois ser criados os vários programas nas várias linguagens, usando os editores criados para o efeito.

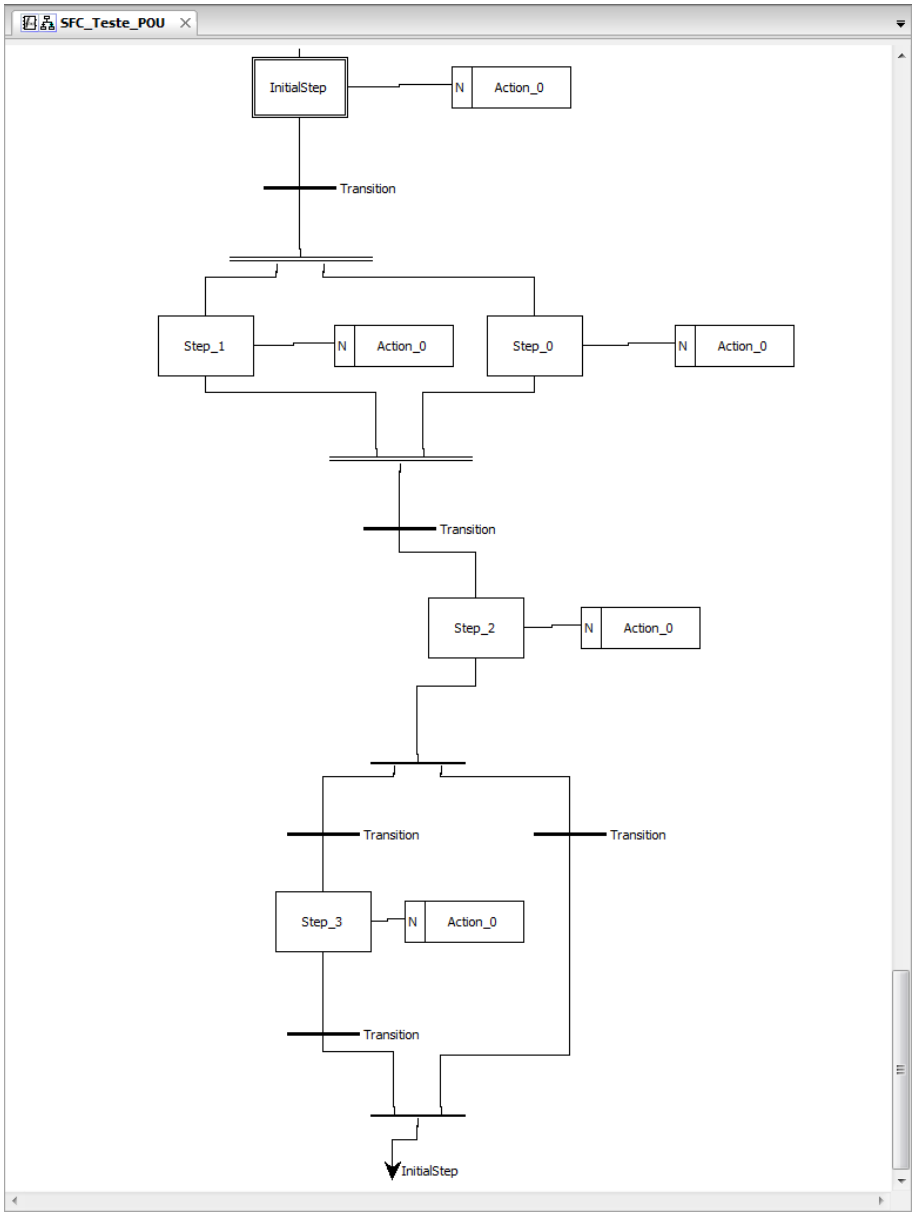


Figura 4.8: Diagrama SFC aberto pelo Beremiz.

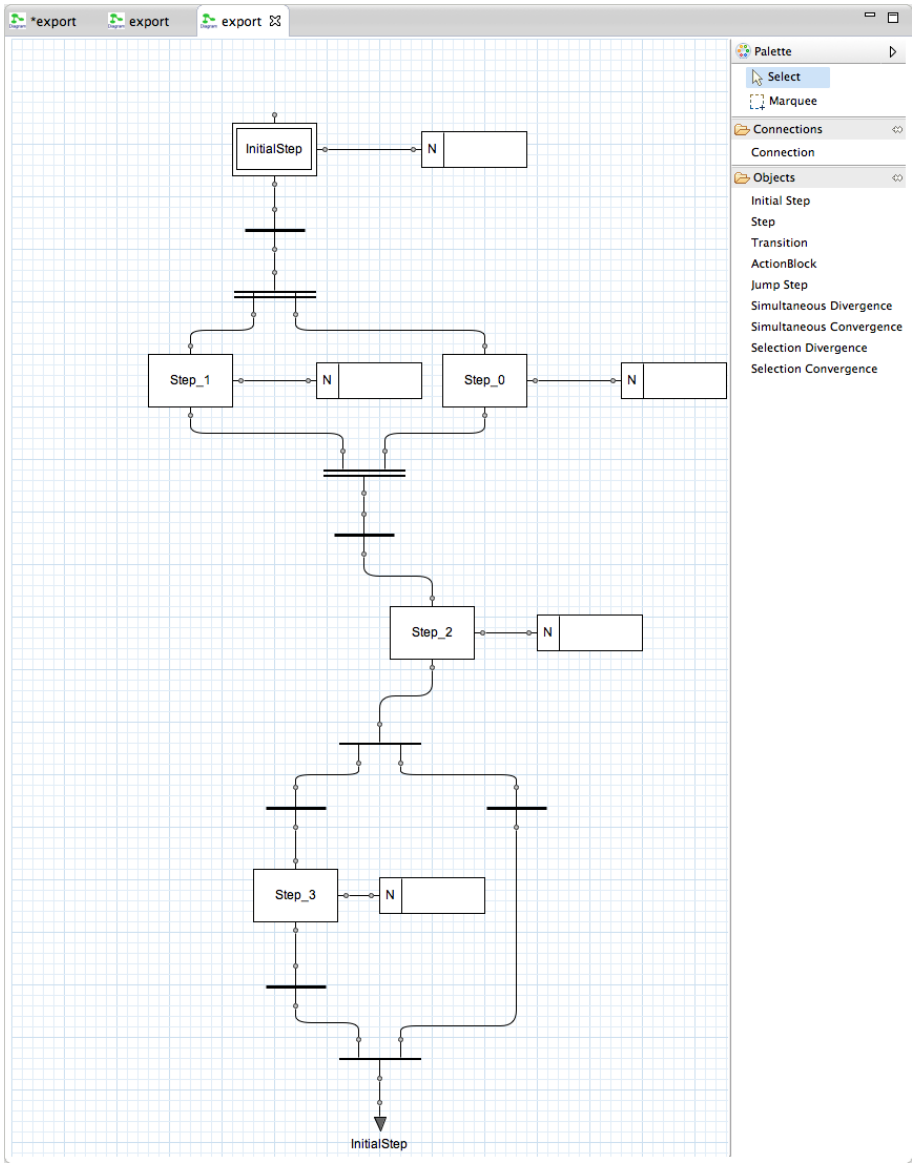


Figura 4.9: Um dos diagrama SFC utilizado para ser aberto por outros IDE's.

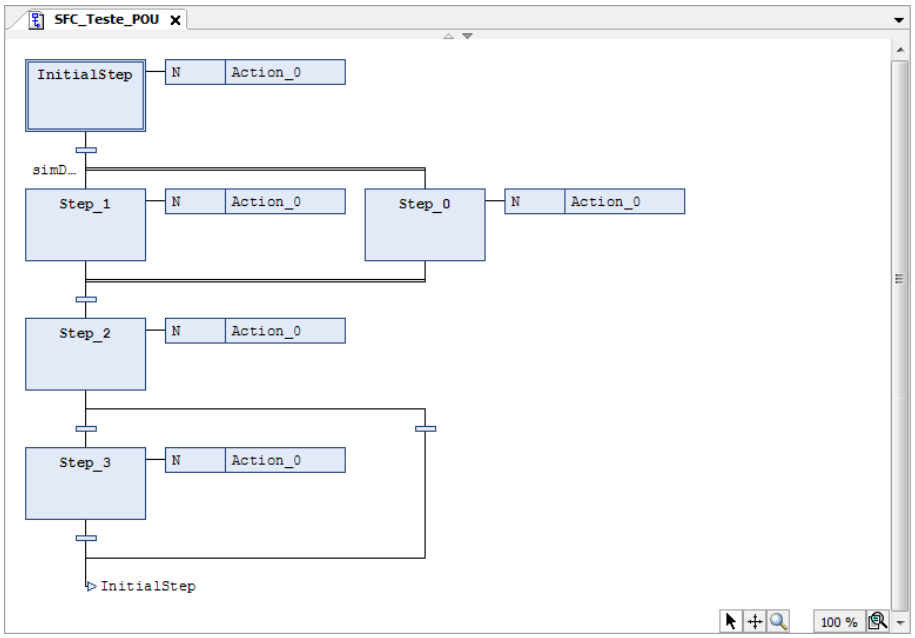


Figura 4.10: Diagrama SFC importado pelo CoDeSys.

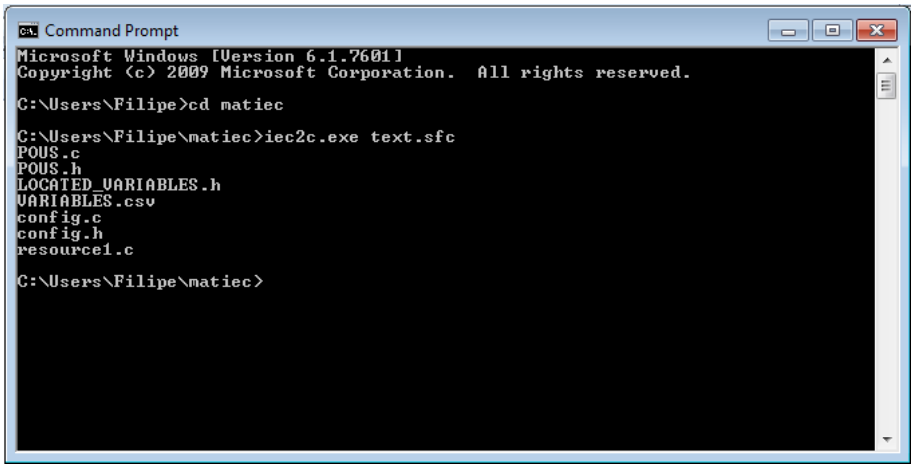


Figura 4.11: Resultado da compilação MATIEC.

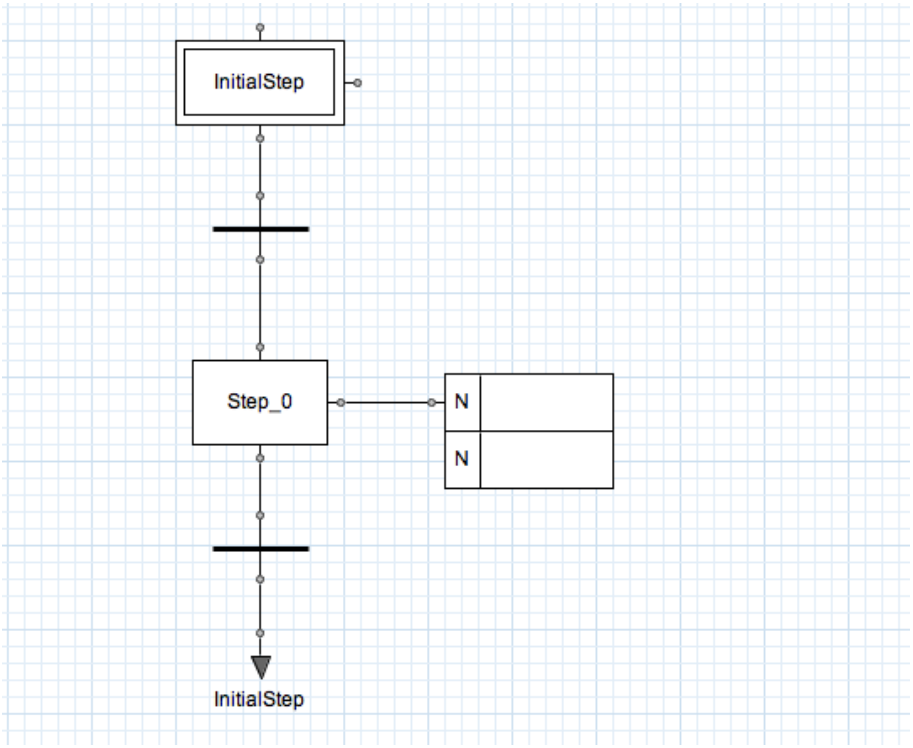


Figura 4.12: Diagrama SFC exportado para texto.

```
1  FUNCTION_BLOCK functionBlock0
2      VAR
3          var1 : INT;
4      END_VAR
5
6      INITIAL_STEP InitialStep:
7      END_STEP
8
9      TRANSITION FROM InitialStep TO Step_0
10         := TRUE;
11     END_TRANSITION
12
13     STEP Step_0:
14         Action_0(N);
15     END_STEP
16
17     ACTION Action_0:
18         var1:=1;
19     END_ACTION
20
21     TRANSITION FROM Step_0 TO InitialStep
22         := TRUE;
23     END_TRANSITION
24
25 END_FUNCTION_BLOCK
26
27 PROGRAM program0
28     VAR
29         FB1 : functionBlock0;
30     END_VAR
31     FB1();
32 END_PROGRAM
33
34 CONFIGURATION config
35     RESOURCE resource1 ON PLC
36         TASK task1(INTERVAL := T#1s,PRIORITY := 0);
37         PROGRAM Prog1 WITH task1 : program0;
38     END_RESOURCE
39 END_CONFIGURATION
40
```

Figura 4.13: Ficheiro de texto gerado pela exportação de um diagrama SFC.

Anexo A

Diagramas de Classes

A.1 Editores Textuais

Pode ver na Figura A.1 o diagrama completo com as classes usadas na criação dos editores textuais para Instruction List e Structured Text.

A.2 Editores Gráficos

Pode ver na Figura A.2 o diagrama completo com as classes usadas na criação do editor gráfico para Sequential Function Chart.

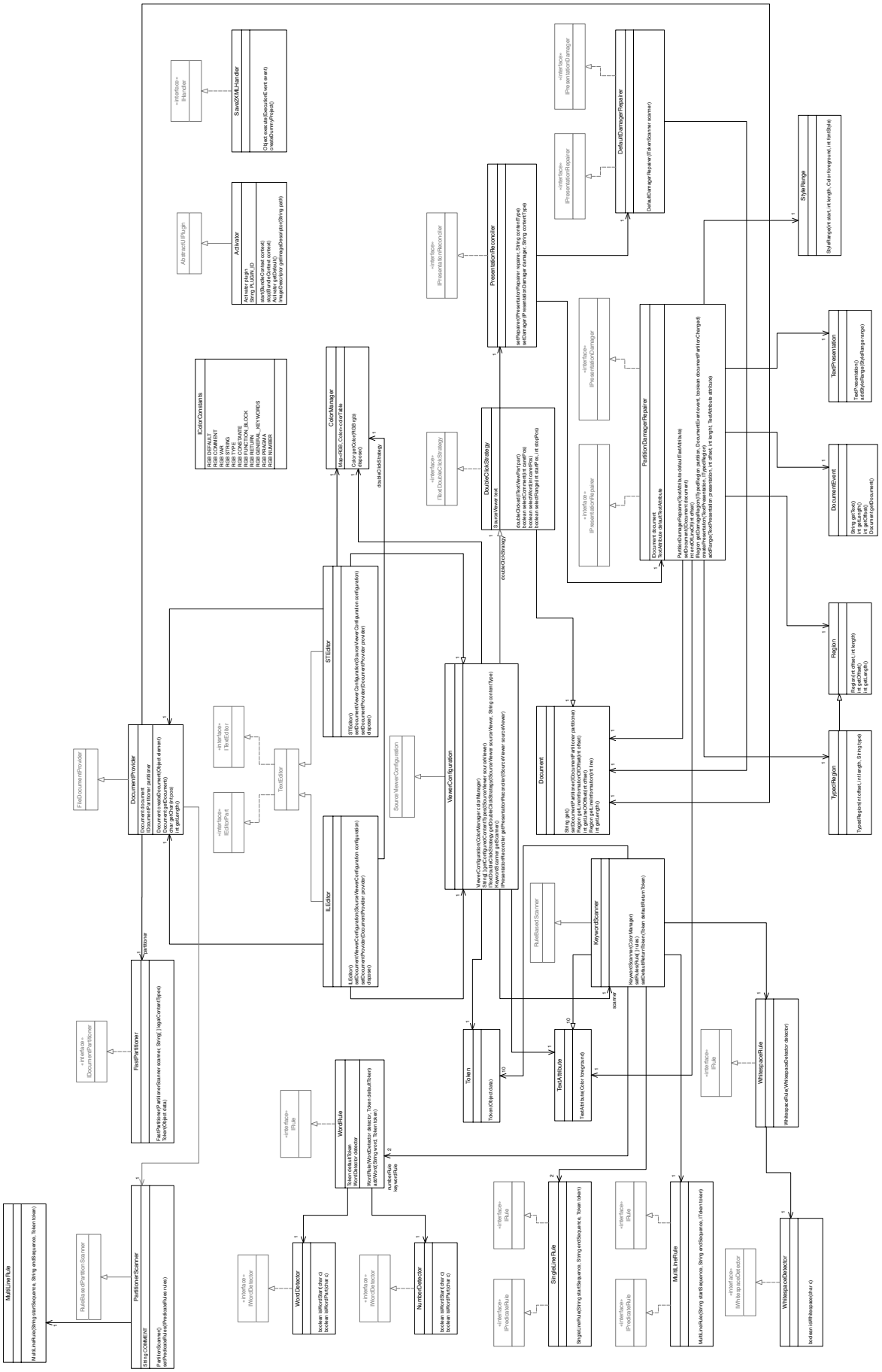


Figura A.1: Diagrama de classes dos Editores Textuais.

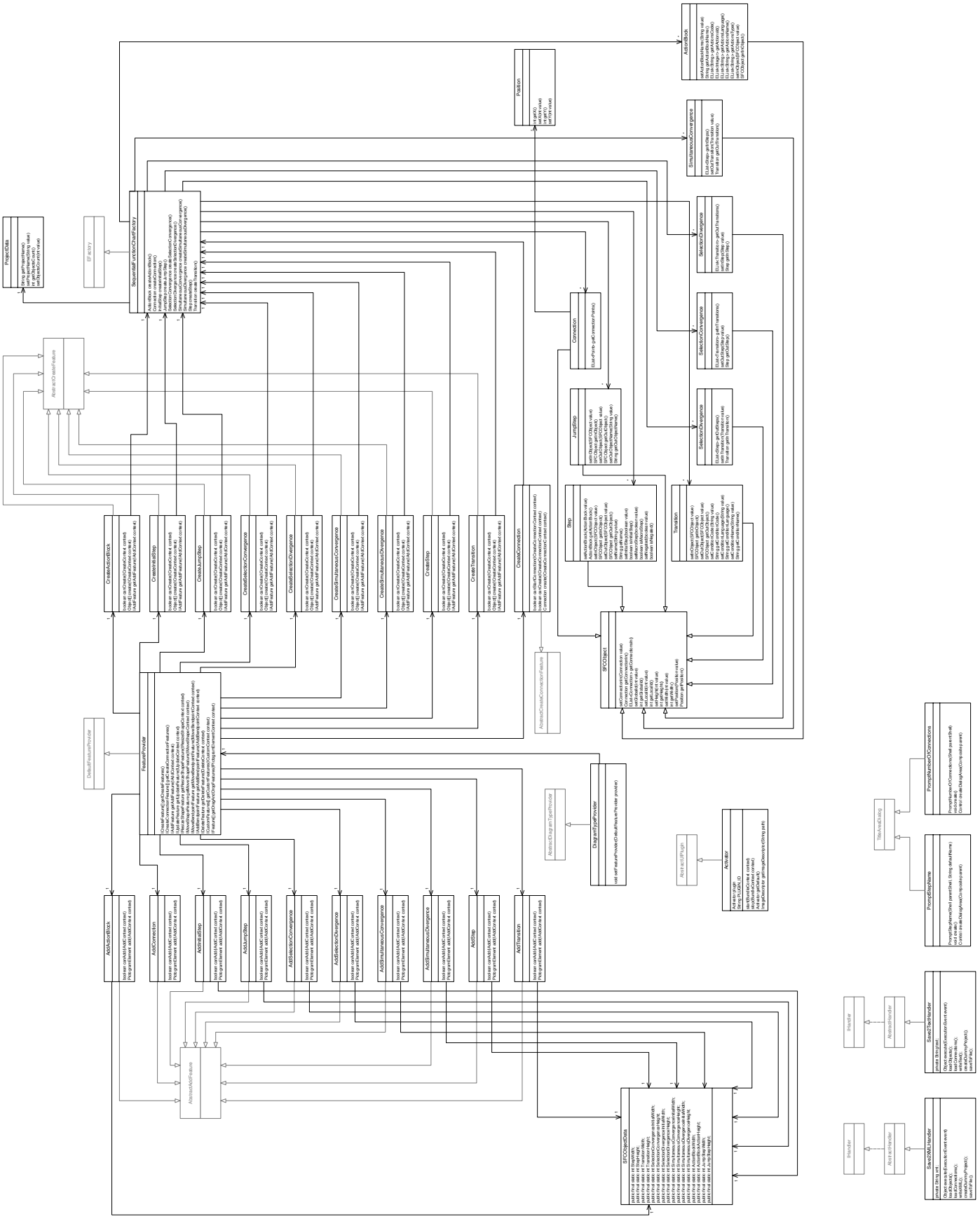


Figura A.2: Diagrama de classes dos Editores Gráficos.

Referências

[1] Antonelli, Pedro Luis. Introdução aos Controladores Lógicos Programáveis (PLCs). Technical report, EJM Engenharia Construção e Comércio Ltda, 1998.

[2] Pupo, Maurício Santos. *Interface homem-máquina para supervisão de um CLP em controle de processos através de WWW*. PhD thesis, Escola de Engenharia de São Carlos da Universidade de São Paulo, 2002.

[3] Wikimedia Commons. CPU 416-3 from series Siemens Simatic S7-400, 2014. URL `http://commons.wikimedia.org/wiki/File:Siemens_Simatic_S7-416-3.jpg`.

[4] IEC. Programmable controllers: Part 3: Programming languages. *IEC Standard P-IEC 61131-3, 2ed.*, 2003.

[5] John, Karl Heinz and Tiegelkamp, Michael. *IEC 61131-3: Programming Industrial Automation Systems: Concepts and Programming Languages, Requirements for Programming Systems, Decision-Making Aids*. Springer, 2001. ISBN 9783642120145.

[6] Tisserant, E and Bessard, L and Sousa, Mário de. An open source IEC 61131-3 integrated development environment. In *5th IEEE International Conference on Industrial Informatics*, pages 183 – 7, Piscataway, NJ, USA, 2007.

[7] Sousa, Mário de. Substation Automation Systems - Overview of IEC 61131-3. Presentation at the Efacec Academy, 2011.

[8] Guimarães, Hugo Casati Ferreira. Norma IEC 61131-3 para programação de controladores programáveis: estudo e aplicação. *Projeto de Graduação. Universidade Federal do Espírito Santo*, 2005.

[9] Shenzhen Youkong Electromechanical. PLC Diagram, 2014. URL `http://www.automation-drive.com/plc-diagram`.

[10] Wikimedia Commons. Turnstile State Machine, 2014. URL `http://commons.wikimedia.org/wiki/File:Turnstile_state_machine_colored.svg`.

[11] Beckhoff Information System. Sequential Function Chart (SFC), 2014. URL `http://infosys.beckhoff.com/english.php?content=../content/1033/tcplccontrol/html/tcplcctrl_languages%20sfc.htm`.

[12] Beremiz. Beremiz, 2014. URL `http://www.beremiz.org`.

[13] 4DIAC. 4DIAC - Open Source for Distributed Industrial Automation, 2014. URL `http://www.fordiac.org`.

[14] Christensen, James. IEC 61499 Function Block Standard: Launch and Takeoff, 2014. URL http://www.automation-drive.com/plc-diagram.

[15] 3S-Smart Software Solutions. CoDeSys, 2014. URL http://www.codesys.com.

[16] Rockwell Automation. ISaGRAF, 2014. URL http://www.isagraf.com.

[17] Khan, Irfa. SAMA Diagrams, 2014. URL http://automation-renew.blogspot.pt/2013/01/sama-diagrams-ii.html.

[18] Schneider Electric. Unity Pro - IEC Programming Software for Modicon PACs, 2014. URL http://www.schneider-electric.com/products/ww/en/3900-pac-plc-other-controllers/3950-pacs/548-unity-pro/.

[19] The Eclipse Foundation. Eclipse, 2014. URL https://www.eclipse.org/home/index.php.

[20] Medeiros, Bruno Dinis Ormonde. Creating IDEs for the Eclipse Platform. *Introdução à Investigação survey*, 2007.

[21] Ho, Elwin. Creating a text-based editor for Eclipse. 2003.

[22] The Eclipse Foundation. GEF Developer Guide, 2014. URL http://help.eclipse.org/luna/nav/24.

[23] The Eclipse Foundation. Graphiti Developer Guide, 2012. URL http://help.eclipse.org/juno/index.jsp?nav=%2F30.

[24] The Eclipse Foundation. Graphiti Proposal, 2010. URL https://www.eclipse.org/proposals/graphiti/.

[25] Beremiz. Beremiz Documentation, 2014. URL http://www.beremiz.org/doc.

[26] Scarpino, Matthew and Good, Nathan A. Build an Eclipse development environment for Perl, Python, and PHP, 2011. URL http://www.ibm.com/developerworks/opensource/tutorials/os-eclipse-octave/os-eclipse-octave-pdf.pdf.

[27] IBM DeveloperWorks. Create a commercial-quality Eclipse IDE, Part 1, 2 and 3, 2006. URL http://www.ibm.com/developerworks/views/opensource/libraryview.jsp?search_by=Create+commercial-quality+eclipse+ide.

[28] The Eclipse Foundation. Plug-in Development Environment Guide, 2012. URL http://help.eclipse.org/juno/index.jsp?nav=%2F4.

[29] The Eclipse Foundation. Workbench User Guide, 2012. URL http://help.eclipse.org/juno/index.jsp?nav=%2F0.

[30] Sousa, Mário de. MATIEC - IEC 61131-3 compiler, 2014. URL https://bitbucket.org/mjsousa/matiec.